JavaLog: un Lenguaje para la Programación de Agentes

Alejandro Zunino, Luis Berdún, Analía Amandi

ISISTAN Research Institute, Facultad de Ciencias Exactas,
Universidad Nacional del Centro de la Pcia. de Buenos Aires
Campus Paraje Arroyo Seco - (B7001BBO) Tandil - Bs. As., Argentina
email: {azunino,lberdun,amandi }@exa.unicen.edu.ar

Resumen

La programación de sistemas multi-agentes ha sido generalmente soportada por lenguajes orientados a objetos o lenguajes lógicos. Ambos paradigmas muestran poseer características para soportar parcialmente el desarrollo de agentes. Sin embargo, si ambos paradigmas son integrados, una solución a la programación de agentes aparece en forma evidente. En este artículo, un lenguaje multi-paradigma para la programación de agentes denominado JavaLog es presentado. Este lenguaje integra el lenguaje orientado a objetos Java y el lenguaje lógico Prolog. Esta combinación permite que agentes sean construidos como objetos manipulando un estado mental definido a través de cláusulas lógicas que son encapsuladas en módulos lógicos. Estos módulos lógicos permiten combinar dinámicamente actitudes mentales para adaptar el comportamiento de agentes considerando diferentes contextos o circunstancias.

Palabras clave: Agentes, Programación Orientada a Agentes.

1. Introducción

La programación de agentes involucra tanto el encapsulamiento de sus comportamientos como el de su estado mental. Estas características nos han llevado a utilizar lenguajes orientados a objetos para programar sistemas multi-agentes.

Los lenguajes orientados a objetos han mostrado poseer varias de las características necesarias para codificar funciones clásicas de agentes. Sin embargo, cuando estados mentales complejos deben ser administrados, la programación lógica ha mostrado ser una mejor alternativa para la implementación de actitudes mentales.

En consecuencia, viejas ideas con relación a lenguajes multi-paradigmas (Ishikawa, 1986) (Fukunaga, 1986) (Mello, 1987) aparecen como una posible solución a la programación de agentes.

Entre las propuestas existentes, sólo aquellas ubicadas en el contexto de utilizar cláusulas lógicas para representar conocimiento interno de objetos hemos utilizado, intentando resolver los inconvenientes encontrados en la programación de agentes y sistemas multi-agentes. En esta línea de trabajo, se propone JavaLog, un lenguaje multi-paradigma que permite construir agentes a partir de objetos Java capaces de manipular conocimiento en formato de cláusulas lógicas Prolog o extensiones de este lenguaje lógico.

La integración de programación orientada a objetos y programación lógica alcanzada en JavaLog ha sido basada en el concepto de módulo lógico. Un módulo lógico es la encapsulación de una secuencia de cláusulas lógicas. Estos módulos pueden ser ubicados tanto en métodos Java como en variables, para luego ser combinados de diversas maneras.

En este artículo se presenta el lenguaje JavaLog

mostrando ejemplos en su utilización para manipular estados mentales de agentes de interfaz.

Por claridad, el artículo se organiza de la siguiente manera. La sección 2 presenta los componentes más relevantes del lenguaje. La sección 3 presenta un ejemplo de manipulación de módulos lógicos. La sección 4 presenta algunos aspectos sobre integración de paradigmas. La sección 5 describe algunos trabajos relacionados. En la sección 6 se presentan experiencias en el uso de JavaLog. Finalmente, las conclusiones son expuestas.

2. JavaLog

JavaLog es un lenguaje de programación que combina los paradigmas de orientación a objetos y lógicos a través de la utilización de Java y Prolog. En el proceso de integrar estos lenguajes para facilitar la programación de agentes se ha desarrollado un interprete Prolog en el lenguaje Java con el fin de posibilitar extensiones del mismo a través de sub-clasificación.

Programar agentes con JavaLog es programar un agente como un objeto Java, el cual es instancia de una clase que representa ese tipo de agente. La funcionalidad del agente es implementada en métodos codificados básicamente en Java.

Módulos lógicos compuestos por una secuencia de cláusulas lógicas Prolog son también utilizados en la programación de los agentes. Conocimiento privado a un agente es ubicado en módulos lógicos referenciados por variables de instancia de los agentes. Conocimiento común a los agentes de una clase es ubicado en módulos lógicos que pueden localizarse en los propios métodos de la clase o referenciados por variables accesibles por todos los objetos de la clase.

Las clases que definan algún tipo de agente se asociarán a una clase denominada *Brain* que permite que cada instancia de esas clases genere una instancia de este *Brain* que representa una instancia del intérprete Prolog. En otras palabras, cada objeto-agente tendrá asociado un objeto *Brain* que le permitirá manipular cláusulas lógicas.

La base de conocimiento de una instancia del intérprete Prolog de un agente cualquiera estará inicialmente vacía. Los módulos lógicos definidos en variables o métodos de la clase no están ubicados en esta base de conocimiento. Los módulos lógicos referenciados por variables tienen que ser explícitamente agregados o retirados en la base de conocimiento. Los módulos lógicos localizados entre el código Java de métodos serán trasladados temporalmente a la base de conocimiento cuando estos métodos sean invocados y mientras estén siendo ejecutados.

3. Módulos lógicos

Como hemos mencionado, el lenguaje JavaLog permite definir módulos lógicos dentro de métodos y variables. De esta manera, un agente puede ser definido como un objeto con conocimiento representado como cláusulas lógicas, de manera tal de aprovechar mejor las características de la programación orientada a objetos y la programación lógica para construir agentes. La presente sección describe a través de un ejemplo, las diversas formas de utilización de módulos lógicos aprovechando las características de integración con Java.

Con el fin de ejemplificar las diversas formas de utilizar módulos lógicos se describirá un agente asistente personal. Dicho agente tiene la función de organizar los horarios de un usuario teniendo en cuenta sus preferencias, tales como tipos de actividades, horarios, lugares, amistades, etc.

En las siguientes secciones se asume que los objetos en los cuales se utiliza la integración Prolog-Java poseen una variable de instancia *brain*, la cual referencia una instancia del intérprete JavaLog, encargado de almacenar y manipular los estados mentales de los agentes. Se denominará *objeto-agente* a un objeto con las propiedades antes mencionadas, es decir, capaz de representar y manipular estados mentales.

En las siguientes sub-secciones se analizan partes del agente asistente, exponiendo los principales aspectos de integración de paradigmas lógico y de orientación a objetos.

3.1 Módulos lógicos en variables y métodos Java

La utilización de módulos lógicos en JavaLog puede realizarse tanto en métodos como en variables. Para clarificar la utilización de módulos lógicos por variables, la Figura 1 muestra el diagrama de un agente representado por un objeto-agente de la clase *AsistentePersonal*. Dicha clase posee tres variables de instancia *contexto1*, *contexto2* y *contexto3*, referenciando cada una de ellas un módulo lógico con diversas preferencias de un usuario sobre eventos sociales, de negocios, deportes, etc. Por ejemplo, la variable *contexto1* define para un usuario que la realización de un evento de negocios en el cual participe su jefe tiene una preferencia con valor 10 y que un evento de golf tiene una preferencia valorada en 9.

Es importante destacar que aunque un objeto-agente defina varios módulos lógicos, esto no significa que utilice todos en un instante determinado. El lenguaje permite a un objeto-agente decidir qué módulos lógicos utilizar. Es decir que cuando se realiza una consulta sobre los estados mentales de un agente, sólo se utilizan los módulos lógicos activos en ese momento para resolver dicha consulta, mientras que los módulos lógicos inactivos son ignorados.

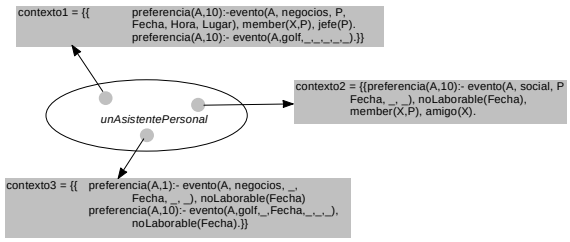

Figura 1. Módulos lógicos en variables de instancia

Este mecanismo es utilizado por el programador en forma directa, es decir, indicando qué módulos son utilizados en un determinado punto de la ejecución del agente. Esto es realizado colocando el siguiente código Java: *contexto1.enable()*.

De esta manera, uno o más módulos lógicos pueden ser habilitados u activados. La habilitación de módulos implica que estos módulos son ubicados en la base de conocimiento del intérprete Prolog del agente que los habilita, considerando estrictamente el orden de las cláusulas según el orden de activación. Así, cuando un agente realiza una consulta, por ejemplo a través del código *?- preferencia(X,9)*, se utilizan sólo las cláusulas de los módulos lógicos habilitados en la base de conocimiento del agente que realiza la consulta.

Módulos lógicos también pueden ser definidos en métodos, expresando así conocimiento común de los agentes de la clase. La figura 2 expone un ejemplo en el cual el módulo lógico referenciado por la variable *contexto1* es habilitado y seguidamente otro módulo sería habilitado. Este segundo módulo está explícitamente definido en el método Java mediante cláusulas Prolog colocadas entre llaves. En este módulo lógico se define que un evento tiene una preferencia 10 si éste es relativo a negocios en el exterior. Este último módulo es activado, localizándose en la base de conocimiento del objeto-agente receptor del mensaje, cuando es invocado el método que lo contiene.

En la misma figura 2, un segundo módulo lógico entre llaves es definido. Éste contiene sólo una consulta que es realizada en el momento que se ejecute el método que la contiene. El resultado de la consulta queda referenciado por el variable local *X*, la cual es accesible por el sector de código Java de ese método.

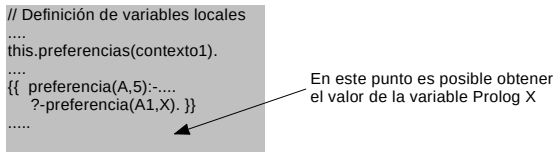

Figura 2. Módulos lógicos en métodos Java

Las reglas de ámbito del lenguaje establecen que, en un punto en donde se efectúa una consulta se utilizan todos los módulos lógicos definidos hasta el momento (mediante llaves dobles) según las reglas de ámbito de Java y los módulos activados explícitamente. Así, por ejemplo, en la consulta de la Figura 2 se utiliza el módulo *contexto1* y el módulo entre llaves.

Luego de efectuada la consulta, los módulos lógicos utilizados en ese momento pueden ser desactivados. Por defecto, una vez que un método terminó su ejecución los módulos activados por éste son eliminados de la base de conocimiento del objeto-agente receptor del mensaje.

La deshabilitación explícita de módulos lógicos es permitida si estos están referenciados por variables. Así, el código *contexto1.disable()* elimina el módulo lógico *contexto1* de la base de conocimiento del agente que ejecuta este código.

Los mecanismos de integración ejemplificados muestran algunas de las posibilidades del lenguaje para manipular estados mentales representados como cláusulas lógicas por parte de un objeto-agente. A continuación se describen las operaciones para combinar módulos lógicos.

3.2. Operando con módulos lógicos

JavaLog permite combinar módulos lógicos según las operaciones definidas en (O'Keefe, 1985). El lenguaje define las siguientes operaciones:

- redefinición: la operación "*a* rewrite *b*", siendo *a* y *b* módulos lógicos, denota un módulo lógico que contiene todas las cláusulas definidas en *b* añadidas a las de *a* cuyo nombre de cabeza no es el mismo que para alguna

cláusula de *b*.

- adición: la operación "*a* add *b*", siendo *a* y *b* módulos lógicos denota un módulo lógico que contiene todas las cláusulas del módulo *a* y a continuación las del módulo *b*.

JavaLog permite utilizar estas operaciones con módulos definidos tanto en variables como en métodos.

Con el fin de ejemplificar la combinación de módulos lógicos referenciados por variables, considérese el asistente personal anteriormente descripto. Dicho agente posee tres variables de instancia *contexto1*, *contexto2* y *contexto3* conteniendo las preferencias de un usuario. Así, por ejemplo *contexto1.add(contexto2)* resulta en un módulo lógico conteniendo las siguientes cláusulas:

```
preferencia(A,10):-
        evento(A,negocios,P,Fecha,Hora,Lugar),
        member(X,P), jefe(P).
preferencia(A,9):-
        evento(A,golf,_,_,_,_).
preferencia(A,10):-
        evento(A,social,P,Fecha,_,_),
        noLaborable(Fecha), member(X,P),
        amigo(X).
```

contexto2.add(contexto1) resulta en un módulo lógico conteniendo las siguientes cláusulas:

```
preferencia(A,10):-
        evento(A,social,P,Fecha,_,_),
        noLaborable(Fecha), member(X,P),
        amigo(X).
preferencia(A,10):-
        evento(A,negocios,P,Fecha,Hora,Lugar),
        member(X,P), jefe(P).
preferencia(A,9):-
        evento(A,golf,_,_,_,_).
```

contexto1.rewrite(contexto2) resulta en un módulo lógico conteniendo las siguientes cláusulas:

```
preferencia(A,10):-
        evento(A,negocios,P,Fecha,Hora,Lugar),
        member(X,P), jefe(P).
preferencia(A,9):-
        evento(A,golf,_,_,_,_).
```

Para combinar módulos lógicos localizados en métodos se utiliza el concepto de herencia de la programación orientada a objetos. Considérese la clase *AsistentePersonal* presentada en la figura 3. El método *evaluarPreferencias* se utiliza para, dado un evento, obtener el valor numérico de dicho evento que representa la valoración del mismo según las preferencias del usuario.

La clase *AsistentePersonalTrabajo* redefine el método *evaluarPreferencias* para dar mayor prioridad a los eventos relacionados con actividades laborales. En este caso, las cláusulas de la superclase son redefinidas mediante el operador *rewrite*. Nótese que el módulo lógico de la subclase está delimitado por {% y %}. Esto denota la operación *rewrite* del módulo lógico de la subclase con el de la superclase.

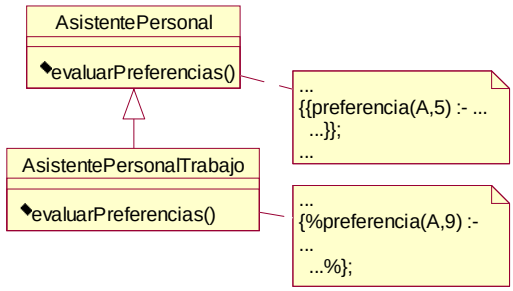

Figura 3. Redefinición de un módulo lógico

Considérese otro escenario, en el cual no se desean redefinir las cláusulas presentes en la superclase, sino que se necesitan añadir cláusulas que especifican la forma de tratar eventos relacionados con el trabajo. En la Figura 4 se ejemplifica este caso.

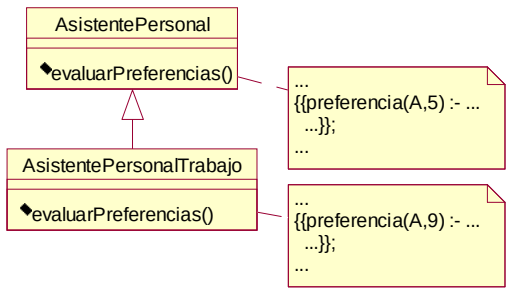

Figura 4. Adición de módulos lógicos

En esta sección se ejemplificó la utilización de módulos lógicos. La siguiente sección trata acerca de la integración de los paradigmas al nivel de objetos y términos.

4. Integración al nivel de objetos y términos

Uno de las mayores dificultades en la integración de los paradigmas de objetos y lógico, lo constituyen las diferencias de ambos en lo referido a las construcciones que cada uno de ellos es capaz de manipular. Así, por ejemplo, el paradigma de objetos se limita a tratar con objetos y mensajes entre objetos. Por otro lado, la programación en lógica utiliza cláusulas y términos.

Para que la integración de los paradigmas sea efectiva, es necesario que el lenguaje defina mecanismos que posibiliten la manipulación de ambas construcciones desde cualquiera de los dos lenguajes.

Con el fin de lograr tales objetivos, el lenguaje JavaLog permite:

1. Convertir objetos en términos.
2. Utilizar objetos en cláusulas como si fuesen términos Prolog.
3. Manipular objetos disponibles en el ámbito de un módulo lógico.
4. Manipular términos lógicos utilizados en un método desde la parte no lógica (o sea, desde código Java) del método.

La idea de convertir objetos en términos lógicos es la de permitir manipular objetos Java desde un programa Prolog. Considérese, por ejemplo, un objeto de la clase *Fecha*. Dicho objeto posee tres variables de instancia: *día*, *mes* y *año*. En la Figura 5 se muestra el término equivalente. El término que representa a un objeto tiene el mismo nombre que la clase del objeto que representa. El primer argumento es una referencia al objeto, de forma tal de poder acceder al objeto a partir del término. El resto de los argumentos son las variables de instancia del objeto.

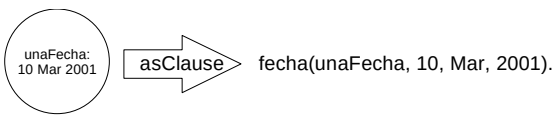

Figura 5. Objetos como términos

JavaLog permite manipular objetos, es decir, enviarles mensajes, de forma tal de permitir tanto el acceso a sus datos, como a su comportamiento. Para enviarle un mensaje a un objeto se utiliza la cláusula *send*. Por ejemplo, *send(unaFecha, toString, [], Str)* instancia *Str* con una representación textual del objeto *unaFecha*. Este efecto se obtiene a partir del envío del mensaje *toString* al objeto *unaFecha*. El tercer argumento de *send* ([]) especifica que el mensaje *toString* no tiene argumentos.

El ítem 3 de nuestra lista se refiere a la posibilidad de utilizar variables Java en un módulo lógico perteneciente al mismo ámbito, como si fuese una variable Prolog. Por ejemplo, el método de la Figura 6 se invoca cuando el asistente personal recibe un nuevo evento. Dicho método define un módulo lógico con los datos del evento recibido para luego ser evaluado de acuerdo a las preferencias del usuario. Para poder utilizar una variable Java dentro de un módulo lógico, ésta debe estar delimitada por #.

Así, en nuestro ejemplo, la variable Java nominada *e* es utilizada en una cláusula Prolog dentro del módulo lógico en este caso definido en el mismo método.

```
nuevoEvento( Fecha f, Vector participantes,
Hora hora, Lugar lugar, Asunto asunto ) {
...
Evento e = new Evento( Asunto, .... );
....
{{ evento(#e#, #Asunto#,
#participantes#..... }};
....
}
```

Figura 6. Variables Java en módulos lógicos

Finalmente, el lenguaje permite manipular los términos utilizados en un módulo lógico de un método desde el ámbito de un método Java. Esto permite, por ejemplo, obtener valores de variables utilizadas en una consulta Prolog desde Java.

5. Trabajos Relacionados

Varios lenguajes han sido propuestos para la programación de agentes (Kellett, 2001) (Denti, 1999) (Fisher, 1994) (Poggy, 1994) (Weerasooriya, 1995). Algunos de ellos utilizan conceptos del paradigma de orientación a objetos en un contexto lógico. Por ejemplo, el lenguage Metatem (Kellett, 2001) (Fisher, 1994) está basado en lógica temporal, encapsulando un conjunto de reglas.

La mayoría de los lenguajes para la programación de agente (por ejemplo, (Poggy, 1994) (Weerasooriya, 1995) (Denti, 1999)) están soportados por conceptos de programación orientada a objetos sin considerar en absoluto los fundamentos lógicos de la especificación de estados mentales. Esta carencia podría ser solucionada utilizando alguna arquitectura que materialice estas

ideas en términos de objetos, pagando de esta manera el costo de manipular manualmente las relaciones entre actitudes mentales.

A diferencia de los trabajos mencionados, nuestro enfoque intenta tomar ventaja de ambos paradigmas a partir de una integración de ambos estilos de programación.

6. Experiencias

Varias experiencias han sido realizadas con este lenguaje. Por ejemplo, un agente de interfaz para generar periódicos personalizados en Internet denominado NewsAgent (Cordero, 1999) es una de estas experiencias.

NewsAgent es un agente que aprende las preferencias de los usuarios observándolos cuando ellos están leyendo noticias en periódicos localizados en la WWW. A partir de este aprendizaje, el agente genera diariamente un periódico personal para cada usuario respetando sus preferencias, ahorrándole así tiempo en la examinación de noticias que no son de su interés y colocando prioritariamente aquellas de mayor interés.

7. Conclusiones

Se ha presentado en este artículo el lenguaje de programación JavaLog que permite la implementeación de sistemas multi-agentes utilizando tanto Java como Prolog. Un agente es definido como un objeto Java, el cual manipula su estado mental a través de programación lógica ya que sus actitudes mentales están definidas por cláusulas lógicas.

Nuestras experiencias han mostrado su utilidad en la simplificación de manipulación de estados mentales en sistemas multi-agentes implementados básicamente con objetos.

Referencias

D. Cordero, P. Roldan, S. Schiaffino, A. Amandi. (1999) “Intelligent Agents Generating Personal Newspapers”. In Proceedings of the International Conference on Enterprise Information Systems, Portugal.

E. Denti, A. Omicini, (1999) “Engineering Multi-Agent Systems in LuCe” in Proceedings of the Workshop on Multi-Agent Systems in Logic Programming - MAS’99 (in conjunction with the International Conference on Logic Programming 1999), Las Cruces, New Mexico, USA, December 4, 1999.

M. Fisher. (1994) “Representing and Executing Agent-Based Systems”. In Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures, and Languages

K. Fukunaga, S. Hirose. (1986) “An Experience with a Prolog-Based Object-Oriented Language. Proc. OOPSLA´86 Conference.

Y. Ishikawa, M. Tokoro. (1986) “A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: It’s features and implementation”. SIGPLAN Notices, 21(11): 232-241.

A. Kellett, M. Fisher (2001) “Coordinating Heterogeneous Components Using Executable Temporal Logic”. In, Meyer and Treur (eds), Agents, Reasoning and Dynamics. Vol. 6 in Series of Handbooks in Defeasible Reasoning and Uncertainty Management Systems. Kluwer Academic Publishers.

P. Mello, A. Natali. (1987) “Objects as Communicating Prolog Units”. In Proceedings of ECCOP’87, European Conference on Object-Oriented Programming.

R. O'Keefe. (1985) “Towards an Algebra for Constructing Logic Programs”. In J. Cohen and J. Conery (eds), Proceedings of IEEE Symposium on Logic Programming, IEEE Computer Society Press, New York, pages 152-160, 1985.

A. Poggy. (1994) “Daisy: an Object-Oriented System for Distributed Artificial Intelligence”. In Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures, and Languages.

D. Weerasooriya, A. Rao, K. Ramamohanarao. (1995) “Design of a Concurrent Agent-Oriented Language”. In Wooldridge, M.; Jennings, N. (Eds.). Intelligent Agents (LNAI 890).