JavaLog: una integración de objetos y lógica para la programación de agentes

Ramiro Iturregui*, Alejandro Zunino, Analía Amandi

Instituto ISISTAN, Facultad de Ciencias Exactas,
Universidad Nacional de Centro de la Pcia de Buenos Aires.
Campus Paraje Arroyo Seco - (7000) Tandil - Bs. As., Argentina
email: {riturr,azunino,amandi}@exa.unicen.edu.ar

Resumen

Los lenguajes orientados a objetos son utilizados en la programación de agentes y, generalmente, satisfacen muchos de sus requerimientos pero presentan inconvenientes en la representación y tratamiento de actitudes mentales. Los lenguajes lógicos permiten representar actitudes mentales en forma declarativa pero no ofrecen la posibilidad de encapsular y ocultar cláusulas lógicas. Con esto, un lenguaje lógico no es un buen candidato para representar las acciones de los agentes.
Una integración de los paradigmas orientado a objetos y lógico puede aportar las ventajas de ambos para el modelamiento de agentes. En este artículo se presenta una integración entre los lenguajes Java y Prolog y la definición de un lenguaje para la programación de agentes y sistemas multi-agente.

1 Introducción

La programación orientada a agentes se ha introducido como una especialización de la programación orientada a objetos [Shoham, 1993]. Los lenguajes orientados a objetos poseen características que permiten satisfacer parte de los requerimientos de la programación orientada a agentes. Un agente puede modelarse mediante un objeto en el cual los métodos representan sus habilidades y las variables de instancia su estado mental. De esta forma, un objeto encapsulará en sus métodos capacidades comportamentales de un agente y su conocimiento estará dado por el estado de sus variables de instancia. Sin embargo, los lenguajes orientados a objetos muestran limitaciones a la hora de tratar con las actitudes mentales de los agentes debido a que se deben implementar distintos algoritmos de inferencia sobre las variables de instancia. Estos algoritmos son, generalmente, costosos de implementar y poco flexibles a cambios.

Por otro lado, los lenguajes lógicos permiten representar actitudes mentales en forma declarativa por medio de cláusulas lógicas. Estas cláusulas lógicas son interpretadas por algoritmos deductivos que, en la programación de agentes, dan origen a razonamientos dependientes del conocimiento propio de los agentes. Los lenguajes lógicos presentan deficiencias en la programación de agentes debido a su imposibilidad de encapsular información y tratar con conocimiento privado.

*Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

En este sentido, la programación orientada a agentes se asociada con dos paradigmas [Amandi et al., 1998]: la programación orientada a objetos y la programación lógica. Las propuestas de modularizar los lenguajes lógicos (CPU [Mello and Natali, 1987], SPOOL [Fukunaga and Hirose, 1986], LOO [Marcarella et al., 1995], SCOOP [Vaucher et al., 1988]) muestran diferentes alternativas para incorporar modularidad desde el punto de vista de la programación orientada a objetos. Estos lenguajes, en su mayoría, definen una clase como un conjunto de cláusulas donde cada una de estas cláusulas representa un método. La herencia puede ser utilizada desde tres puntos de vista:

- *agregación*: la definición de una cláusula con una misma cabeza y aridad en una subclase implica la agregación de esta cláusula a la definición del método de la superclase. Una subclase puede completar la definición de su superclase pero no puede redefinir cláusulas.
- *redefinición parcial*: los métodos están definidos por un conjunto de cláusulas. Una cláusula con igual cabeza y aridad dentro de un método del mismo nombre que en su superclase redefine la cláusula correspondiente. Una subclase redefine cláusulas con igual cabeza y aridad, pero no puede completar una definición de su superclase.
- *redefinición total*: la definición de un método con el mismo nombre que en la superclase implica la redefinición completa del mismo método. Una subclase redefine un método completo con lo cual no puede reutilizar la definición en su superclase.

Por otro lado, las propuestas de manipular cláusulas lógicas en lenguajes orientados a objetos (ORIENT/84 [Ishikawa and Tokoro, 1986]) utilizan variables de instancia para almacenar cláusulas. De esta forma permiten la utilización de bases de conocimiento en objetos por medio de herramientas apropiadas.

Cada una de estas propuestas presenta una aplicación diferente del conjunto de cláusulas que representa el conocimiento o el comportamiento en la programación orientada a agentes. Ninguna de estas propuestas posibilitan la utilización conjunta de cláusulas para definir conocimiento y como parte de las habilidades de los agentes.

La utilización de una integración de ambos paradigmas ofrece la posibilidad de aprovechar las características principales de cada uno en la representación de conocimiento y habilidades. Es decir, la incorporación de lógica en la programación orientada a objetos permite representar el estado mental de los agentes en forma declarativa, mientras que los aspectos comportamentales quedan expresados como métodos de la programación orientada a objetos donde es posible utilizar cláusulas lógicas en la toma de desiciones.

En este trabajo se presenta una integración entre un lenguaje orientado a objetos, Java, y un lenguaje lógico, Prolog. Esta integración se ha llamado JavaLog y provee distintas formas de integrar objetos y lógica para la programación de agentes.

En la siguiente sección (2) se incorporan las bases teóricas necesarias para una integración entre los paradigmas lógico y orientado a objetos. Luego, en la sección 3, se detalla la definición y utilización del lenguaje de programación de agentes JavaLog. En la sección 4 se describen las primitivas de comunicación. Por último se presentan algunos trabajos relacionados (5)y las conclusiones (6).

2 Integración de objetos y lógica

Los paradigmas lógico y orientado a objetos trabajan sobre diferentes elementos. Para integrar la programación orientada a objetos con la programación lógica y viceversa es necesario realizar un mapeo entre los elementos de cada uno.

La programación orientada a objetos utiliza elementos (objetos) que son accedidos en un contexto restringido. Estos objetos son manipulados por métodos y encapsulan datos propios de la entidad que modelan. Por otro lado, la programación lógica trabaja con términos y cláusulas.

JavaLog es una integración entre los paradigmas lógico y orientado a objetos desarrollado para la programación de agentes. La programación orientada a agentes utiliza objetos para modelar las habilidades de los agentes y cláusulas lógicas para modelar el estado mental de dichos agentes. Las habilidades de los agentes pueden estar representadas por capacidades de acción como por capacidades de comunicación con otros agentes, mientras que el estado mental puede presentarse en forma de creencias, intensiones, compromisos [Cohen and Levesque, 1990] , etc. Esta integración define el módulo como el componente básico de manipulación [Amandi et al., 1999]. En el dominio de JavaLog, un objeto y un método de la programación orientada a objetos se ven como módulos. Un objeto es un módulo que encapsula datos y un método es un módulo que encapsula comportamiento. Además, JavaLog utiliza el concepto de módulo lógico como un conjunto de cláusulas de Horn [O'Keefe, 1985]. De esta manera, un agente puede modelarse como una composición de módulos.

En la figura 1 se pueden observar dos agentes compuestos por módulos y una referencia a un objeto (llamado brain) que hace las veces de cerebro del agente. Este objeto es una instancia de un intérprete de módulos lógicos que utiliza cláusulas Prolog. Cada agente es una instancia de una clase que puede definir parte de sus métodos en Java y parte en Prolog. Así, la definición de los métodos de una clase puede estar compuesta por varios módulos lógicos. Una instancia de una clase de este tipo presenta una composición de módulos lógicos definidos en métodos y módulos lógicos referenciados por variables de instancia.

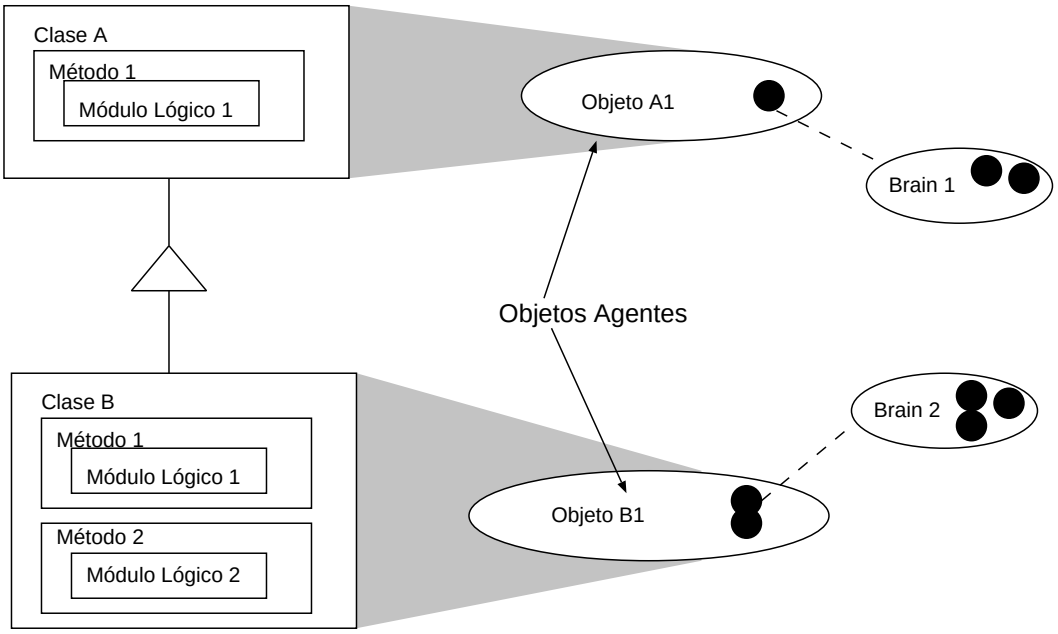

Figura 1: Esquema de composición de módulos

Para realizar esta composición de módulos lógicos y orientados a objetos mediante la integración de objetos y lógica, JavaLog provee un intérprete Prolog basado en el concepto de módulo. Este intérprete está implementado en Java y permite utilizar objetos dentro de cláusulas lógicas, así como embeber módulos lógicos dentro de código Java. En las secciones siguientes se describen las distintas formas de integrar objetos y lógica con JavaLog.

3 El lenguaje JavaLog

Para poder utilizar objetos Java en programas Prolog y módulos lógicos Prolog en objetos Java es necesario contar con un conjunto de primitivas que permitan esta integración. JavaLog ofrece un lenguaje de interacción objetos-lógica-objetos que le brinda al programador de agentes la posibilidad de utilizar las características de ambos paradigmas.

3.1 Objetos Java en programas Prolog

En la programación de agentes se requiere manipular un conjunto de estados mentales que son resultado, o dan como resultado, la utilización de algunas de las habilidades del agente. Para modelar los estados mentales como cláusulas lógicas y las habilidades como métodos de objetos es necesario poseer una forma de interacción que permita:

- Crear objetos Java en programas Prolog
- Adquirir una representación lógica de un objeto creado en Java
- Enviar mensajes a objetos en evaluaciones Prolog.

Creación

Cuando en Prolog se desea construir algún tipo de visualización, por ejemplo, por razones de claridad y eficiencia, resulta conveniente utilizar las herramientas de visualización que provee un lenguaje orientado a objetos como Java. Así, si se desea crear un botón en una ventana desde un programa Prolog, se puede utilizar el predicado predefinido `newInstance( +Class, +Var)` que invoca al constructor de la clase `Class` y coloca el objeto creado en `Var` de la forma:

`newInstance( 'Button', Button1).`

Esta primitiva presenta una de las formas de incorporar objetos Java en los programas Prolog. De esta manera, se pueden crear objetos que luego serán utilizados e inclusive almacenados en una base de datos Prolog.

Interacción Java-Prolog

Una de las características de mayor utilidad es la posibilidad de representar objetos Java en cláusulas Prolog. Es decir, es posible obtener una representación en forma de cláusula lógica un objeto. Esto se logra utilizando un mensaje particular llamado `toClause` de la forma:

```
PlClause.toClause(aPerson, new UnaryFunction[ ] {
                  new UnaryFunction() {
                      public Object execute(Object arg) {
                             return ((Person)arg).firstname() }
                      }
                  new UnaryFunction() {
                      public Object Execute(Object arg) {
                             return ((Person)arg).lastname() }
                      }
                  } );
```

Este fragmento de código toma un objeto `aPerson`, instancia de la clase `Person` y lo transforma en una cláusula Prolog, instancia de la clase `PlClause`. El método `toClause` ejecuta las funciones de un arreglo (bloque) pasado como segundo argumento sobre el primer argumento. Este arreglo contiene la política de mapeo del objeto a la cláusula. Es decir, en este ejemplo se desea representar al objeto `aPerson` con una estructura con functor `person` (nombre de la clase) y argumentos nombre y apellido (de acuerdo al estado del objeto).

Supongamos que este fragmento de código se ejecuta con el objeto `aPerson` cuyo valor de retorno del mensaje `firstname()` es `Francisco` y el de `lastname()` es `Sanchez`, la cláusula correspondiente será

```
person( Aperson, 'Francisco', 'Sanchez').
```

Donde `Aperson` es una variable ligada con el objeto `aPerson`.

Envío de mensajes

Una vez que se tiene una forma de acceder a un objeto Java, es necesario poder acceder al estado interno del mismo, u obtener algún comportamiento particular. En la programación orientada a objetos esto se logra con el envío de mensajes al objeto destino. Este proceso. en la programación JavaLog , se llama invocación de módulos (en este caso de comportamiento).

JavaLog provee un predicado predefinido que, dado un objeto (módulo que encapsula datos), invoca sus métodos (módulos de comportamiento). Por ejemplo, si con una consulta Prolog se recupera el módulo que encapsula los datos de Francisco Sanchez:

```
?- person( X, 'Francisco', 'Sanchez').
```

La variable `X` se ligará con la instancia de `Person` correspondiente. Si se desea conocer la edad de Francisco Sanchez, es posible invocar al método `age()`:

```
?- send( X, age, [ ], Y).
```

Esta consulta enviará el mensaje `age` al objeto `aPerson` y ligará la variable `Y` con el valor retornado por el método correspondiente. Esto es equivalente a la sentencia Java:

```
Y = aPerson.age();
```

3.2 Prolog Embebido

La característica más importante de JavaLog es, tal vez, la posibilidad de manipular en forma indistinta módulos lógicos y módulos orientados a objetos. Esta característica guía la definición del lenguaje JavaLog en base al manejo de módulos lógicos. Un módulo lógico puede almacenarse en una base de datos Prolog o puede asignarse a variables Java de forma tal que pueda ser activado cuando se necesite utilizar su contenido en una evaluación determinada. De esta forma, las cláusulas lógicas embebidas en los métodos Java se traducen a módulos lógicos asociados a la clase del objeto mediante el preprocesador JavaLog. El preprocesador JavaLog toma como entrada código Java con Prolog embebido (código JavaLog) y genera un archivo con el código Java equivalente.

Módulos lógicos

La interacción entre módulos lógicos y módulos orientados a objetos esta definida en base a tres parámetros: referencia, comunicación y composición.

Cuando se habla de referencia, se establece la locación de los módulos lógicos como variables de instancia o como parte de los métodos de las clases. De esta manera un objeto JavaLog puede contener conocimiento privado en cláusulas lógicas.

Las capacidades de acción de los objetos-agentes JavaLog vienen dadas por los métodos Java que tiene la posibilidad de encapsular módulos lógicos y consultas sobre una base lógica. Por ejemplo, supongamos un agente comerciante como una instancia de la clase *CommerceAgent* que tiene la habilidad de seleccionar y comprar elementos de acuerdo a las preferencias de la persona que representa. Las preferencias de este agente pueden cambiar por razones contextuales, con lo cual, la habilidad de comprar cada tipo de elemento está regida por las preferencias activas. Por esta razón, los distintos conjuntos de preferencias se almacenan en módulos lógicos como el que sigue.

```
preference(car, [ford, Model, Price]):-
           Model > 1998,
           Price < 60000.
preference(motorcycle, [yamaha, Model, Price]):-
           Model >= 1999,
           Price < 9000.
```

Utilizando JavaLog, la clase del agente comerciante se define de la siguiente manera:

```
public class CommerceAgent {
       private Brain brain;
       private PlLogicModule userPreferences;

       public CommerceAgent( PlLogicModule userPreferences ) {
              brain = new Brain()
              this.userPreferences = userPreferences;
       }
       ...
       public Brain brain(){ return this.brain; }
       ...
       public boolean buyArticle( Article anArticle ) {
              userPreferences.enable();
              type = anArticle.type;
              if (?-preferences( #anArticle#, [#type#,
                    #brand#, #model#, #price#]).)
                  buy(anArticle);
              userPreferences.disable();
       }
}
```

En este ejemplo se puede observar que la variable `userPreferences` referencia un módulo lógico con las preferencias del agente. Estas preferencias son habilitadas cuando se necesitan (por ejemplo en la decisión de la compra o no del artículo) y son deshabilitadas después de utilizarse. Para

verificar la aceptabilidad de la compra, se embebe una consulta Prolog que utiliza el conocimiento activado del agente acerca de sus preferencias. Esta consulta contiene variables encerradas ente `#`. Esta es la forma en que JavaLog utiliza las variables Java en consultas Prolog.

Cuando se tiene un agente de este tipo se debe definir un método `brain()` que retorna una instancia del intérprete JavaLog. El intérprete JavaLog es el encargado de realizar todas las deducciones a partir de módulos lógicos. Este intérprete es asociado con lo que sería el cerebro del objeto-agente JavaLog.

Ahora bien, si se necesita construir un agente que cambie de preferencias de acuerdo a ciertas condiciones dadas en su contexto, es posible definir un módulo lógico para cada una de las condiciones. Es decir, suponiendo que el agente comerciante posee las preferencias anteriores en condiciones normales, pero ante una necesidad de inversión puede cambiar sus preferencias a:

```
preference(car, [ _, Model, Price]):-
            Model > 1995,
            Price < 100000.
preference(motorcycle, [ _, Model, Price]):-
            Model >= 1995,
            Price < 15000.
```

De esta manera, el agente comerciante puede contener dos o más variables de instancia con las distintas preferencias. Estas preferencias serán cambiadas cuando el usuario del agente lo desee utilizando el método `switchContext`. El siguiente fragmento de código muestra esta incorporación a la clase `CommerceAgent`.

```
public class CommerceAgent {
       private Brain brain;
       private int actualPref;
       private PlLogicModule[ ] userPreferences;

       public CommerceAgent( PlLogicModule userPreferences ) {
              brain = new Brain()
              this.userPreferences[0] = userPreferences;
              this.actualPref = 0;
       }
       ...
       public Brain brain(){ return this.brain; }
       ...
       public int addPreferences( PlLogicModule userPreferences ) {
              this.userPreferences[this.userPreferences.length + 1] =
                   userPreferences;
              return this.userPreferences.length + 1;
       }
       ...
       public void switchContext( int newContext ) {
              this.actualPref =ñewContext;
       }
       ...
       public boolean buyArticle( Article anArticle ) {
              userPreferences[actualPref].enable();
              type = anArticle.type;
              if (?-preferences( #anArticle#, [#type#,
                    #brand#, #model#, #price#]).)
                  buy(anArticle);
              userPreferences[actualPref].disable();
       }
}
```

En un objeto de la clase `CommerceAgent` se pueden incorporar la cantidad de módulos lógicos con preferencias que se deseen. Para conocer qué preferencia tiene que utilizar en la decisión de una compra, el agente comerciante conoce cuál es el módulo lógico que debe activar en el momento de la compra por medio de una variable de instancia `actualPref`.

Otra implementación de esta clase podría contemplar la posibilidad de realizar combinaciones de módulos lógicos de acuerdo a ciertas condiciones variables, con lo cual las posibilidades de utilización del conocimiento del agente se ven ampliamente incrementadas.

El manejo de los módulos lógicos (también llamados capacidades del agente) puede ser delegado al intérprete JavaLog. Una instancia de la clase `Brain` puede manipular una cantidad de capacidades de un agente con la utilización de primitivas de agregado, activación y desactivación de capacidades. De esta manera, la clase `CommerceAgent` podría codificarse de la siguiente manera:

```
public class CommerceAgent {
       private Brain brain;
       private int nextContext;

       public CommerceAgent( String userPreferences ) {
              brain = new Brain()
              brain().addCapability("Context1", userPreferences);
              nextContext=2;
       }
       ...
       public Brain brain(){ return this.brain; }
       ...
       public int addPreferences( String userPreferences ) {
              brain().addCapability("Context" + nextContext,
                                    userPreferences);
              return nextContext++;
       }
       ...
       public void switchContext( int newContext ) {
              brain().deleteCapabilities();
              brain().activeCapability( "Context" + newContext);
       }
       ...
       public boolean buyArticle( Article anArticle ) {
              type = anArticle.type;
              if (?-preferences( #anArticle#, [#type#,
                    #brand#, #model#, #price#]).)
                  buy(anArticle);
       }
}
```

En este ejemplo se puede observar la utilización de los métodos para el manejo de las capacidades de un agente. El método `addCapability` agrega un módulo lógico al conocimiento del agente, `activeCapability` activa para la evaluación un módulo lógico y `deleteCapabilies` desactiva todos los módulos lógicos del agente.

Por otra parte, JavaLog permite embeber módulos lógicos en módulos de comportamiento (métodos). Por ejemplo podría crearse una instancia del agente comerciante anteriormente definido de la siguiente manera:

```
CommerceAgent anAgent = new ComerceAgent(
    {{ preference(car, [ford, Model, Price]):-
           Model > 1998,
           Price < 60000.
       preference(motorcycle, [yamaha, Model, Price]):-
           Model >= 1999,
           Price < 9000.
    }} );
```

Donde el código encerrado entre {{ y }} es traducido a un módulo lógico que es pasado como parámetro al constructor de la clase `CommerceAgent`.

Cuando se embeben módulos lógicos en métodos, JavaLog aplica una política de activación de dichos módulos tal que, un módulo puede ser activado explícita o implícitamente y puede ser desactivado manual o automáticamente. El ejemplo anterior muestra una forma de activación/desactivación explícita. El siguiente código muestra un ejemplo de activación implícita y desactivación automática.

```
...
public void extra() {
     {{ edge(4,6). }};
}
...
public void test() {
     Integer i = new Integer( 1 );
     extra();
     System.out.println( "edge(X,Y) " + ?-edge(X,Y).
                         + " X=" + brain().answer().get("X")
                         + " Y=" + brain().answer().get("Y") );
     {{ edge(#i#,2).
        edge(2,3).
        edge(3,4).
        edge(6,7).
        connected(X,Y):-edge(X,Y).
        connected(X,Y):-edge(X,Z),edge(Z,Y).
        ?-edge(1,Y).
        ?-connected(A,B).
        ?-connected(4,7).
     }};
     System.out.println( brain().answer() );
}
```

Este código muestra la inclusión de dos módulos lógicos dentro de métodos Java. El código Prolog encerrado entre doble llaves en los métodos `extra()` y `test()` se traduce a módulos lógicos incluidos en las capacidades del `brain` de un objeto de esta clase. La invocación al método `extra()` en `test()` produce la activación del módulo lógico asociado con ese método. Luego, una consulta Prolog del tipo `?-edge(X, Y)` producirá una evaluación sobre el módulo lógico `extra`. Seguidamente se activará el módulo lógico asociado con el método `test()`, el cual incorporará una serie de cláusulas a su base de conocimientos y luego tratará de satisfacer las consultas correspondientes. Una vez realizadas estas tres consultas (consultas múltiples), el intérprete `brain` desactivará todos sus módulos lógicos (incluyendo `extra`).

Resumiendo, la inclusión de consultas dentro de módulos lógicos embebidos en métodos Java produce la desactivación automática de todas las capacidades activas del agente, mientras que una consulta embebida en el código Java, se evalúa sin efectos colaterales.

JavaLog brinda una forma de acceder al valor al cual quedaron ligadas las variables Prolog después de una consulta, así como el resultado de las últimas consultas (en el caso de consultas múltiples). Para acceder al valor de una variable puede utilizarse un diccionario de variables conteniendo pares del tipo *(nombre, valor)*. En el ejemplo anterior, este proceso está resumido

en `brain.answer().get("X")`. Para recuperar el valor de la *i-ésima* consulta se podrá utilizar `brain.result( i )`.

Composición de Módulo lógicos

La composición de módulos lógicos puede ser tratada desde dos puntos de vista.

- Los módulos lógicos referenciados por variables pueden ser combinados con la utilización de métodos predefinidos. Por ejemplo `enable()` y `disable()`en el caso del agente comerciante pueden utilizarse para activar cada uno de los conocimientos en forma separada o combinada.
- Los módulos lógicos embebidos en métodos Java pueden ser combinados mediante la herencia.

Así, por ejemplo el siguiente fragmento de código muestra un método definido con código JavaLog. El sector de código delimitado entre "`{{`" y "`}}`" define la forma de evaluación de un estudiante en forma de cláusulas Prolog (un módulo lógico). A continuación, el código contiene una consulta Prolog que obtiene la calificación de un estudiante. En este caso el predicado Prolog incluye una variable Java `#aStudent#` con un objeto estudiante.

```
public class Profesor {
  private Brain brain;
  ...
  public Brain brain(){ return this.brain; }
  ...
  public String qualification() {
    {{ qualification(Student, 'A'):- exercise(Student, passed),
                                     finalTest(Student, passed).
       qualification(Student, 'B'):- exercise(Student, passed).
       ?- qualification(#aStudent#, X). }};
    if( brain().result(1) )
       return brain().answer().get( "X" ).toString();
    else
       return null;
  }
}
```

Una subclase de `Profesor` puede redefinir el método `qualification`. La forma de realizar dicha redefinición permite obtener distintas combinaciones de módulos lógicos: redefinición o conjunción de módulos lógicos. Si el método de la superclase contiene una consulta en el módulo lógico embebido, toda subclase de esa clase redefinirá el módulo lógico correspondiente. Si, en cambio, el módulo lógico de la superclase no contiene consultas, la subclasificación producirá una agregación de módulos lógicos.

En el caso de la clase `Profesor`, el módulo lógico contenido en el método `qualification( Student aStudent)` incluye una consulta, con lo cual cualquier redefinición del método concluirá en una redefinición del módulo lógico. Por ejemplo, la clase `PostGradoProfesor` redefine este método y utiliza la evaluación sobre el módulo lógico de su superclase.

```
public class PostGradoProfesor extends Profesor {
...
public String qualification( Student aStudent ) {
     String qualification=super().qualification(aStudent)
     {{ qualification(Student, 'A'):-
                  #qualification# = 'A',
                  seminary(Student, 'A').
         qualification(Student, 'B'):-
                  #qualification# = 'A',
                  seminary(Student, 'B').
        ?- qualification(#aStudent#, X). }};
    return brain().answer().get( "X" ).toString();
  ...
}
```

Para conseguir la agregación de módulos lógicos, una clase puede embeber módulos lógicos que sólo contengan cláusulas (sin consultas). En el caso del profesor, si la consulta `?- qualification(#aStudent#, X)` no se encontrara dentro del módulo lógico, cualquier subclase podría agregar un módulo lógico con cláusulas con la misma cabeza, o bien redefinir dichas cláusulas (sin invocar al `super`). El siguiente fragmento de código muestra una nueva definición de la clase `Profesor`.

```
public class Profesor {
  private Brain brain;
  ...
  public Brain brain(){ return this.brain; }
  ...
  public String qualification() {
    {{ qualification(Student, 'A'):- exercise(1, Student, passed),
                                     exercise(2, Student, passed),
                                     finalTest(Student, passed).
       qualification(Student, 'B'):- exercise(Student, passed),
                                     exercise(2, Student, passed).
    }};
  }
  ...
  public String qualification( Student aStudent ) {
    qualification();
    ?- qualification(#aStudent#, X). ;
    if( brain().result(1) )
       return brain().answer().get( "X" ).toString();
    else
       return null;
  }
}
```

Una instancia de esta clase, cuando recibe el mensaje `qualification( aStudent )` invoca el método `qualification()`. Este método agrega el módulo lógico a su base de conocimiento. Luego se realiza la consulta a partir de esta base de conocimiento.

Si se quiere implementar la clase correspondiente a un profesor con otra forma de calificación, se pueden agregar niveles de calificación al Profesor (redefiniendo `qualification()`), como muestra el siguiente ejemplo.

```
public class ProfesorA extends Profesor{
  ...
  public String qualification() {
    super.qualification();
    {{ qualification(Student, 'C'):- exercise(1, Student, passed).
    }};
  }
  ...
}
```

En esta redefinición del método `qualification()` la invocación a `super.qualification()` incorpora a la base de conocimiento el módulo lógico del método de la superclase y luego se agrega el módulo lógico definido en el método de la subclase. Por otro lado, si se omite esta invocación, la base de conocimiento contendrá sólo el módulo lógico definido en la subclase, con lo que se logra el efecto de reescritura del módulo lógico.

Existe otra forma de modificar la base de conocimiento del objeto en forma dinámica. Es decir, la posibilidad de tratar los módulos lógicos y los módulos orientados a objetos sin distinción permite que un módulo lógico sea pasado como parámetro a un método y, de esta manera, ser activado en el cerebro del objeto-agente.

Preprocesador JavaLog

Una vez que el código JavaLog es escrito es necesario traducirlo a su equivalente en código Java. Esta tarea es realizada por el preprocesador JavaLog. El preprocesador JavaLog toma como entrada un archivo de extensión `javalog` y genera un archivo con extensión `java` que luego puede ser compilado a bytecode por cualquier compilador jdk 1.1.3 o superiores.

Si, por ejemplo, el código del agente comerciante se encuentra en un archivo `commerceagent` con extensión `javalog`, la siguiente invocación al preprocesador generará el archivo `commerceagent` con extensión `.java`.

```
./JavaLog.sh -p commerceagent.javalog
```

4 Interacción entre agentes

Uno de los puntos de estudio fundamentales en la programación de agentes es la comunicación en sistemas multiagentes. En un sistema de este tipo existen una cantidad de agentes ocupándose de realizar la tarea que le ha sido asignada. Para esto, es posible que necesiten tanto recursos compartidos, como conocimiento compartido. Es decir, es frecuente que un agente necesite del conocimiento (o parte del conocimiento) de otro agente para alcanzar un objetivo propio [Zunino and Amandi, 1999].

Cuando se trata con sistemas multiagentes, donde estos agentes comparten parte de su conocimiento o tienen objetivos comunes, es necesario contar con alguna forma de organización del sistema para obtener la mayor productividad individual y colectiva. Un primer paso hacia esta organización es la posibilidad de contar con algún medio de comunicación que permita implementar algoritmos de coordinación.

JavaLog incorpora una arquitectura de referencia para la programación de sistemas multiagentes que permite modelar un agente como un objeto Java y su estado mental como cláusulas Prolog. Además, brinda una serie de primitivas de comunicación entre agentes para compartir conocimientos y objetivos. En este aspecto, el concepto de módulo lógico toma fundamental importancia ya que cada agente encapsula parte de su conocimiento (o todo su conocimiento) en módulos lógicos. Estos módulos lógicos pueden ser privados al agente o pueden ser publicados en un repositorio Prolog para que pueda ser utilizado por el resto de la comunidad de agentes. Este repositorio se ajusta a una arquitectura Blackboard [Buschmann et al., 1996], donde se posee un medio de comunicación centralizado en el cual los distintos componentes de la arquitectura leen o escriben datos relevantes al procesamiento de las tareas colectivas. Además, estos agentes pueden comunicarse entre si (sin la utilización del Blackboard) para, por ejemplo, pedir ayuda a otros agentes. De esta manera, un agente que es incapaz de resolver una consulta puede delegar esta tarea, o parte de ella, a otro agente que con su conocimiento, puede llevar a cabo satisfactoriamente la evaluación de la misma consulta.

En las próximas secciones se describirán las primitivas de comunicación que ofrece JavaLog para una arquitectura distribuida y una arquitectura Blackboard.

4.1 Arquitectura Blackboard

La arquitectura Blackboard está definida en función a su componente principal, el blackboard. Un blackboard [Buschmann et al., 1996] es una estructura de datos que puede ser leída y modificada por programas llamados *fuentes de conocimiento* (**K**nowledge **S**ource). Cada fuente de conocimiento se especializa en la resolución de una parte particular de un tarea completa. Así, todas las KS trabajan en forma conjunta en la búsqueda de una solución. Estos programas especializados son independientes entre sí. El rumbo tomado por el sistema está determinado principalmente por el estado del procesamiento, el cual es evaluado por un componente central que controla y coordina los programas especializados. De esta forma, es posible realizar un control de los datos de manera oportunista. Esto aumenta la experiencia del componente de control, a partir de lo cual, puede construir heurísticas derivadas experimentalmente para controlar el procesamiento.

JavaLog ofrece una serie de primitivas de comunicación donde las fuentes de conocimiento (objetos agentes) se pueden comunicar con un componente centralizado. En un sistema multiagente, cada componente de la comunidad de agentes posee su base de conocimiento privada y un cerebro (intérprete Prolog) que la interpreta. Además, la comunidad puede tener una base de conocimiento pública (blackboard) y un agente con un comportamiento especial (componente de control).

La figura 2 muestra una arquitectura blackboard definida por un objeto-agente centralizado (componente de control) que posee una base de conocimiento privada. Este intérprete puede hacer pública una parte de su base de conocimiento (blackboard) para que todos los agentes de la comunidad escriban y lean de ella. Además, cada agente del sistema, posee su propio cerebro (intérprete Prolog) con su respectiva base de conocimiento privada.

En este contexto, el agente de control puede publicar un módulo lógico en el blackboard con la primitiva `publishm(ModuleName)`, donde `ModuleName` es el nombre del módulo que se desea publicar. Un agente comparte su estado mental utilizando los predicados `publish` y `publishm` y pregunta por los módulos públicos de un agente con `registryObjs`:

- `publish(Identificador)`: hace pública la base de datos del agente. La base de datos se publica con el nombre *//<NombreHost>/JavaLog/<Identificador>*.

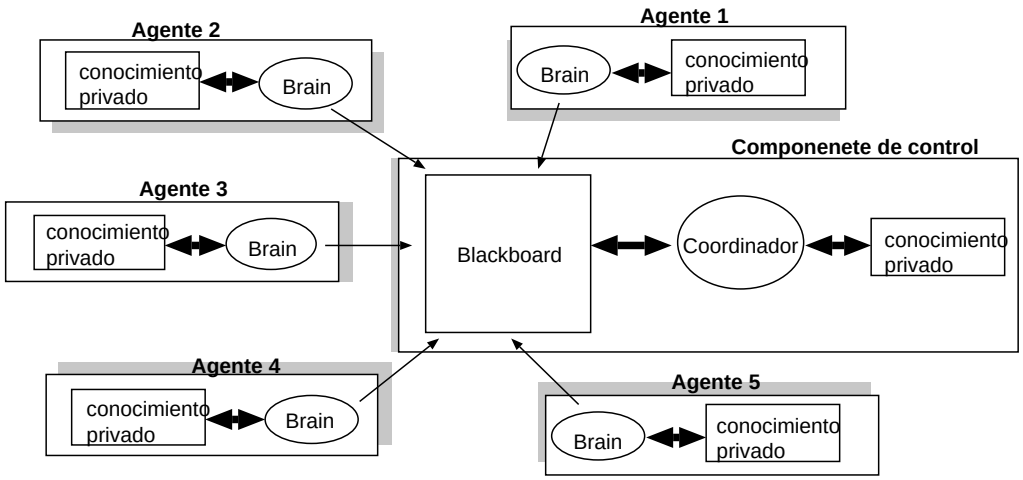

Figura 2: Arquitectura Blackboard

- `publishm(NombreMódulo, Identificador)`: hace público un módulo lógico del agente. El módulo lógico se publica con el nombre *//<NombreHost>/JavaLog/<NombreMódulo>/<Identificador>*.

- `registryObjs(NombreHost, ListaURL)`: consulta con el host `NombreHost` qué módulos lógicos exporta. `ListaURL` se instancia con una lista con los identificadores de módulos remotos.

Luego, otro agente puede conectarse al blackboard utilizando la primitiva `connect(ModuleName)`. El predicado `connect` incorpora una referencia a un módulo lógico remoto en la base de datos local; `connect` recibe un argumento con un identificador de un objeto remoto con la sintaxis *//<NombreHost>/JavaLog/<Identificador>*, donde *NombreHost* es el nombre o la dirección IP (**I**nternet **P**rotocol) del host que contiene el módulo lógico identificado por *Identificador*. De esta forma, la conexión a una base de conocimiento remota puede verse como la incorporación de un módulo lógico *remoto* a la base local . Por ejemplo, si un agente ejecuta:

```
connect('//javaLog.exa.unicen.edu.ar/JavaLog/publicDataBase').
```

se produce la incorporación de un módulo lógico remoto con el contenido de la base de datos del servidor especificado, a la base de datos del agente.

Luego de que un agente se conecta con una base de datos remota, la evaluación de una consulta local se realiza de la misma forma que antes, sólo que ahora se tiene en cuenta el contenido de la base remota. Si este agente desea escribir algo en el blackboard, puede hacerlo con la utilización de `assertm(ModuleName, Clause)`. El predicado `assertm` se utiliza para agregar cláusulas a un módulo lógico, y por lo tanto puede ser usado con un módulo lógico remoto de la misma forma que con un módulo lógico local.

4.2 Delegación

JavaLog provee facilidades para efectuar delegación de tareas entre agentes. El mecanismo de delegación permite a un agente solicitar a otros la realización de tareas determinadas. Dicho mecanismo resulta de utilidad, por ejemplo, cuando existen agentes con diferentes capacidades,

y por consiguiente, con diferente estado mental; así, un agente puede solicitar la colaboración de los agentes aptos para cada tarea en particular. Es decir, un agente puede tener el conocimiento necesario para satisfacer una consulta pero puede carecer de la habilidad de efectuar esa consulta. Por medio del mecanismo de evaluación remota [Fuggetta et al., 1998], un agente puede utilizar el *cerebro* y, por consiguiente, el conocimiento de otro agente para realizar un razonamiento, sin necesidad que el *dueño* del cerebro implemente la habilidad correspondiente

JavaLog define tres predicados para efectuar comunicación directa entre distintos objetos-agente:

- `present(Identificador)`: se utiliza para permitir que otros intérpretes soliciten la evaluación de cláusulas en forma remota. El agente se registra con un identificador con la siguiente sintaxis: *//<NombreHost>/JavaLog/<Identificador>*.
- `rcall(Service, Query)`: evalúa una consulta en el intérprete identificado por `Service`, donde `Service` es un identificador de la forma *//<NombreHost>/JavaLog/<Identificador>*. Si la consulta que se ejecuta en el intérprete remoto es satisfactoria, la evaluación tiene éxito. Si el host no se encuentra, el intérprete remoto no acepta invocaciones remotas, o `Query` no es verdadero, entonces `rcall` falla.
- `rsolutions(Service, Query, Result)`: es similar a `rcall` salvo que solicita al intérprete remoto que considere todas las posibles reevaluaciones de `Query`. Si la evaluación de `Query` tiene éxito, la variable `Result` se instancia con todas las sustituciones posibles que hacen que `Query` sea verdadero.

La utilización de estas primitivas permite la comunicación tanto de agentes integrantes de una misma comunidad, como la comunicación entre agentes de comunidades distintas. Es decir, es posible tener un grupo de agentes compartiendo conocimiento mediante un blackboard donde uno o más de esos agentes tienen la capacidad de comunicarse con agentes externos a su comunidad (que no conocen de la existencia del blackboard). Por medio del envío de mensajes a agentes remotos, este agente externo puede, indirectamente, utilizar el conocimiento del agente con el que se comunica. Este agente, al ser parte de la comunidad del blackboard utiliza en su razonamiento el contenido del mismo. La figura 3 muestra un sistemas de múltiples comunidades de agentes compartiendo conocimiento y la comunicación intercomunidad dada a partir de la comunicación entre pares de agentes.

Así, por ejemplo, la figura 4 muestra al agente comerciante que intenta comprar algunos elementos en un repositorio de artículos. Este repositorio contiene ofertas publicadas por varios agentes vendedores en forma de módulos lógicos. La administración del repositorio está en manos de un agente intermediario. Cuando el agente comerciante se comunica con el coordinador, le realiza una consulta de la forma:

```
rcall( //blackboardManager, article(Type, Offer))
```

Luego, el intermediario ejecuta la consulta en su base de conocimiento (el blackboard) e instancia la variable `Offer` con el resultado de su búsqueda. A partir de este resultado, el agente comerciante lo compara con sus preferencias y decide la compra.

En esta consulta al coordinador, el agente podría utilizar la primitiva `rsolutions`:

```
rsolutions( //blackboard, article(Type, _ ), ListOfOffers)
```

De esta forma, el agente comerciante obtiene un conjunto de artículos que satisfacen la consulta. Luego, debe elegir de acuerdo a sus preferencias, cuál es la mas apropiada.

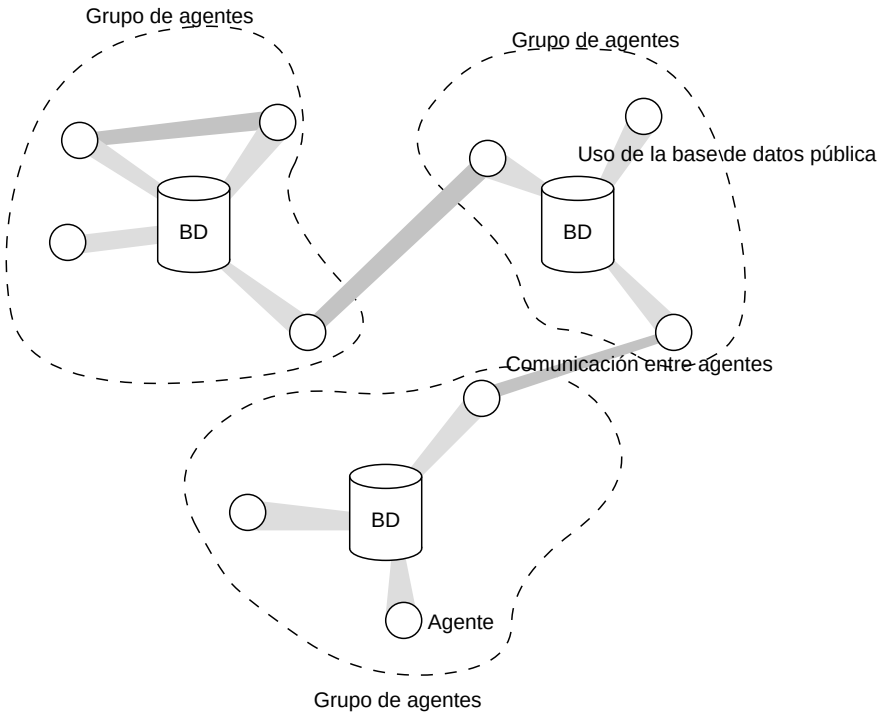

Figura 3: Sistema de múltiples comunidades de agentes

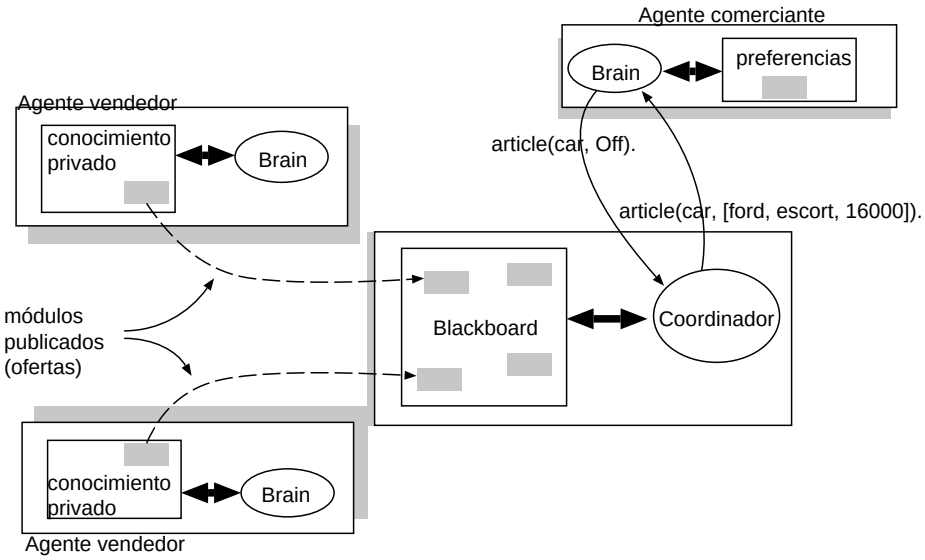

Figura 4: Interacción entre agentes

5 Trabajos relacionados

El lenguaje JavaLog se ha utilizado en diversos proyectos de Inteligencia Artificial. A continuación se describen brevemente algunos de ellos:

- *NewsAgent* [Cordero et al., 1999]: un agente que genera un diario electrónico personalizado a partir de las preferencias recuperadas de un usuario. Este agente utiliza la técnica

de Razonamiento Basado en Casos (CBR [Maher and Pu, 1997, Maher et al., 1995]) para recuperar dichas preferencias a partir de la navegación de diferentes periódicos para representarlas en forma de módulos lógicos.

- *Intelligent Schelude*: agenda inteligente distribuida que representa las preferencias de los usuarios en forma de módulos lógicos y concreta citas en forma autónoma utilizando Razonamiento Basado en Casos.

- *Commerce Agent*: un agente de comercio que busca repositorios con ofertas teniendo como referencias las preferencias y deseos de compra de los usuarios. Utiliza algoritmos de negociación para decidir la compra de productos.

- *Intelligent TEG*: juego de estrategias de guerra que genera planes de acciones y asiste al usuario a partir de estrategias aplicadas anteriormente.

- *GraphPlan* [Blum and Furst, 1997] y *UCPOP* [Penberty and Weld, 1992]: algoritmos de planificación de acciones utilizados por los agentes para construir planes de tareas.

6 Conclusiones y trabajos futuros

En este artículo se ha presentado un lenguaje desde el punto de vista de programación de agentes. Este lenguaje ofrece una combinación multi-paradigma que permite modelar a los agentes como objetos compuestos por distintos tipos de módulos: de datos, de comportamiento y lógicos. En este sentido, el concepto de módulo lógico toma especial importancia a la hora de representar las actitudes mentales de los agentes debido a que permiten la definición de las mismas en forma de cláusulas lógicas.

Se ha introducido una arquitectura de referencia para la programación de sistemas multi-agente. La definición del lenguaje en torno al concepto de módulo lógico facilita la utilización de primitivas de comunicación interagente. En JavaLog, la comunicación entre agentes puede realizarse de dos maneras: comunicación entre pares y mediante un repositorio centralizado.

Por otro lado, la posibilidad de contar con el intérprete de cláusulas lógicas escrito en Java ofrece facilidades en la extensión del mismo. Por ejemplo, es posible incorporar el manejo de lógica temporal o de intensiones mediante la subclasificación de algunas de sus clases.

JavaLog se ha utilizado en diversos desarrollos de sistemas de agentes donde muestra ventajas en el esfuerzo de utilización de actitudes mentales. Actualmente se está desarrollando la versión en Java de BrainstormJ, un framework para la programación de agentes, y herramientas de depuración de programas para asistir al programador de agentes.

Referencias

[Amandi et al., 1998] Amandi, A., Iturregui, R., and Zunino, A. (1998). Object-agent oriented programming. In *Proceedings of ASOO'98 in* 27^{th} *JAIIO*, Buenos Aires, Argentine.

[Amandi et al., 1999] Amandi, A., Zunino, A., and Iturregui, R. (1999). Multi-paradigm languages supporting multi-agent development. In Garijo, F. and Bornar, M., editors, *Multi-Agent System Engineering*, Lecture Notes in Artificial Intelligence, pages 128–139.

[Blum and Furst, 1997] Blum, A. and Furst, M. (1997). Fast planning through planning graph analysis. (90):281–300.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). Blackboard. In *Pattern-Oriented software architecture*, chapter 6, pages 71–95. John Wiley & Sons, England.

[Cohen and Levesque, 1990] Cohen, P. R. and Levesque, H. J. (1990). Intention is choice with commitment. *Artificial Intelligence*, Vol. 42(Num. 2).

[Cordero et al., 1999] Cordero, D., Roldán, P., Schiaffino, S., and Amandi, A. (1999). Intelligent agents generating personal newspapers. In *Proc. of the ICEIS'99*, Setúbal, Portugal.

[Fuggetta et al., 1998] Fuggetta, A., Picco, G. P., and Vigna, G. (1998). Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361.

[Fukunaga and Hirose, 1986] Fukunaga, K. and Hirose, S. (1986). An experience with a prolog-based object-oriented language. In *OOPSLA '86 Conference Proceedings. Sigplan Notices*, pages 224–231.

[Ishikawa and Tokoro, 1986] Ishikawa, Y. and Tokoro, M. (1986). A concurrent object oriented knowledge representation language oriente84/k: it's features and implementation. *SIGPLAN Notices, New York*, Vol. 21(Num. 11):232–241.

[Maher et al., 1995] Maher, M. L., Balachandran, M. B., and Zhang, D. M. (1995). *Case-Based Reasoning in Design*. Lawrence Erlbaum Associates.

[Maher and Pu, 1997] Maher, M. L. and Pu, P. (1997). *Issues and Applications of Case-Based Reasoning in Design*. Lawrence Erlbaum Associates.

[Marcarella et al., 1995] Marcarella, P., Raffaetà, A., and Turini, F. (1995). Loo: An object-oriented logic programming language. *Proc. of Italian Conference on Logic Programming (GULP '95)*.

[Mello and Natali, 1987] Mello, P. and Natali, A. (1987). Objects as communicating prolog units. In *Proceeding of ECOOP'87 European Conference on Object-Oriented Programming*, pages 181–191. Springer-Verlag.

[O'Keefe, 1985] O'Keefe, R. (1985). Towards an algebra for constructing logic programs. In Cohen, J. and Conery, J., editors, *Proceedings of IEEE Symposium on Logic Programming*, pages 152–160, New York. IEEE Computer Society Press.

[Penberty and Weld, 1992] Penberty, J. S. and Weld, D. S. (1992). Ucpop: A sound, complete, partial-order planner for adl. In *Proceedings of KR-92*, pages 103–114, Cambridge, MA.

[Shoham, 1993] Shoham, Y. (1993). Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92.

[Vaucher et al., 1988] Vaucher, J., Lapalme, G., and Malenfant, J. (1988). Scoop, structured concurrent object-oriented prolog. In *Proc. of ECOOP'88 European Conference on Object-Oriented Programming*, pages 191–211. Springer-Verlag.

[Zunino and Amandi, 1999] Zunino, A. and Amandi, A. (1999). Logic modules for communicating distributed agents. In *Workshop de Investigadores en Ciencias de la Computación (WICC'99)*.