Electronic Journal of SADIO

http://www.dc.uba.ar/sadio/ejs

vol. 2, no. 1, pp. 5-16 (1999)

Object-Agent Oriented Programming

Analía Amandi[1] Ramiro Iturregui[1] Alejandro Zunino[1]

[1]Universidad Nacional de Centro de la Provincia de Buenos Aires
Facultad de Ciencias Exactas - ISISTAN
Campus Universitario Paraje Arroyo Seco - (7000) Tandil - Buenos Aires - Argentina
email: {amandi,riturr,azunino}@exa.unicen.edu.ar

Abstract

Object-oriented programming has been used for building intelligent agents, with the limitation it cannot represent complex mental attitudes. With logic programming it is possible to represent and infer relationships among mental attitudes such as intentions, goals and beliefs, with limitations in the usage of capabilities of action.

This paper presents two alternatives for integrating object-oriented with logic programming, which enable agent programming. Java and Smalltalk have been used for providing one typed and another non-typed integration with Prolog.

Keywords: agent-oriented programming, logic programming.

1. Introduction

Agent-oriented programming (AOP) has been presented by Y. Shoham [Shoham, 1993] as a specialization of object-oriented programming (OOP). In this context, objects are considered the base for the design of intelligent agents.

The definition of objects as base for agent design is specially supported by two facts. The first, agents possess a bounded set of action abilities that can be mapped to a set of objects classes methods. The second, agents maintain an internal private state known as mental state, which is equivalent to the internal and private state of the objects.

Far from Shoham's definition, many languages designed for agent programming (i.e. AgentSpeak [Weerasooriya, 1995], Daisy [Poggi, 1995], Metatem [Fisher, 1994], CooL [Kolb, 1995]) have been built using concepts from object-oriented paradigm and many specific agents (such as [Vere, 1990] [Ciancarini, 1997]) have been implemented in object oriented languages such as C++, Smalltalk or Java.

These experiences of agent language definitions based on concepts of object-orientation and the development of multi-agent systems using object-oriented languages put in evidence one limitation in the possibilities for managing mental attitudes. In those experiences, mental attitudes are manipulated as simple data whose relationships are freely interpreted in decision algorithms used by agents.

That limitation is imposed by the complexity of relationships among mental attitudes such as beliefs, goals, preferences, intentions, commitments and possibilities. These relationships among mental attitudes are exposed in several logic formalisms [Cohen, 1990] [Rao, 1991] [Linder, 1996] [Huang, 1996]. In spite of these theoretical developments, complex relationships among mental attitudes are not found in multi-agent developments.

Object-oriented programming resolves the problem that means the hiding of behavior of each type of agents and administration of private knowledge. On the other hand, logic programming allows logic clauses being used for representing mental attitudes.

This paper presents two experiences to integrate object-oriented programming and logic programming as a base for agent development. The next section describes the two basic lines of integration between object-oriented programming and logic programming. Section 3 shows the integration between Java and a Prolog interpreter. Section 4 exposes the integration of Smalltalk and Prolog and an example of agent programming. Finally, at the end of this paper, the conclusions are presented.

2. Integration of Objects and Logic

In the tentative of taking advantages of the modularization and reusability provided by object-oriented languages and of the inference of knowledge provided by logic languages, several alternatives has been analyzed. These alternatives of integration can be characterized in two main lines: extension of logic programming with object-oriented programming concepts and extension of object-oriented programming languages with logic programming concepts.

Extension of logic programming with object-oriented programming concepts

The building of large systems with logic languages presents well-known problems of performance. Furthermore, these systems cannot be reused because of their complexity. For this reason, great efforts have been made for modularizing logic programs.

Several object-oriented languages have been designed to incorporate modularity to logic languages. Generally, these languages have a Prolog-like syntax. As example, the languages CPU [Mello, 1987], SPOOL [Fukunaga, 1986], LOO [Marcarella, 1995] and SCOOP [Vaucher, 1988] can be mentioned. These languages show different alternatives to incorporate modularity in logic programming from the use of object-orientation concepts.

These languages define classes as a set of clauses, where each clause is a method. Inheritance is managed in two different ways for these languages. For introducing these two alternatives, let two classes *A* and *B* (*B* as subclass of *A*) composed by the following methods in form of clauses:

```
Class A:
qualification(Student, 'A') :- passed(Student, finalTest).
passed(Student, finalTest) :- passed(Student, exercise1),
                                   passed(Student, exercise2).
```

```
Class B:
passed(Student), finalTest) :- passed(Student, exercise4).
```

Inheritance is viewed from two points of view. The first considers that clauses in a subclass with the same head that those clauses in the superclass not redefines those methods. In this case, objects *B* use the clauses defined in A more the classes defined in B. This conjunction of clauses for representing inheritance not accept the redefinition of methods.

In the example, an object of class *B* has all the clauses defined in *A* and *B* available. In the example, an object *B* has two ways of considering satisfactory student's final test: when the student passes the exercise 1 and 2, and when he passes the exercise 4.

The second inheritance view is when a clause in a subclass with the same name that those clauses in the superclass redefines those inhered methods. This combination of clauses is one that rewrites clauses with the same name, allowing thus the redefinition of clauses.

In the example, an object of class *B* has all the clauses defined in *B* more the clauses of *A* with head different of the all clause of *B* are available. In the example, an object *B* has one way of considering satisfactory student's final test: when the student passes the exercise 4.

In the first alternative, a subclass can add new clauses with the same name, but it can not redefine clauses; in the second, it is considered the alternative in which the subclass redefine clauses with the same name, but it can not add clauses with the same name.

The examples above show two possibilities of combining logical modules by means of inheritance: the first alternative was adopted by SPOOL [Fukunaga, 1986] and the second by SCOOP [Vaucher, 1988]. Both combinations of logical formulae are useful in the programming of object-oriented applications.

Extension of object oriented programming with concepts of logic programming

The object-oriented programming has certain advantages over other paradigms. These advantages are information hiding, inheritance and modularity. However, in some applications is necessary to manipulate knowledge responding to some kind of logic that logic languages provide. For this reason, the possibility to add knowledge in a declarative form to an object-oriented program became relevant. Examples of languages that integrate knowledge in objects are shown in [Ishikawa, 1986] and [Amaral, 1993]. These languages allow the creation of a knowledge base in each object and the management of it through a set of facilities.

Both of the extensions presented in this paper are in this last category. The reason is that agents behave as objects from an action point of view and internally manage logical relationships for making intelligent decisions.

3. JavaLog: integrating Java and Prolog

JavaLog is an integration of Java and Prolog that allows the resolution of problems using both languages. This capability of interaction between Prolog and Java enable us to take advantageous of the facilities of both paradigms.

This integration has been entirely developed in Java. The development of JavaLog has been made in two stages. In the first stage, a Prolog interpreter was designed and implemented in Java. In the second stage, the machinery that supports the codification of Java methods in Prolog and the use of Java objects in Prolog programs was developed.

The next two subsections present the integration from Java to Prolog and from Prolog to Java.

3.1. Java using Prolog

The possibility of writing Prolog code inside Java programs allows the production of natural solutions to problems that requires logic inference. These problems are common in intelligent agents since the mental attitudes of agents are supported by particular logics.

By means of a preprocessor is possible to embed Prolog into a Java program. JavaLog marks between the strings "{%" and "%}" the Prolog code included in Java methods. For example, the code below shows a Java method that is part of the implementation of an intelligent agent. These intelligent agents generate plans to achieve their goals. Here, the Prolog code between the marks generates an agent plan. A planning algorithm written in Prolog generates the plan. In the example, the characters "#" are used to include Java variables in the Prolog code.

```
LinkGraph links = new LinkGraph(50);
Graph constraints = new Graph(50);
links.initialize( PlList.empty() );
```

```
boolean prologResult;
{% getActions(Domain),
planning(Domain, #links#, #constraints#). %}
```

This integration of objects and logic requires the existence of the following variables in the scope where the embedded Prolog is located:

- A variable named *prologResult* of type *boolean*.
- An instance of the Prolog interpreter in *prolog*.
- All Java variables declared between "#".

Another use of Prolog does not preprocess the code. It consists of the inclusion of atoms with the form *$i* in the Prolog program, where *$i* denotes the *i*-th array element composed of Java objects. When *$i* is used in the Prolog program, the *i*-th array element is taken and it is converted to a Prolog-compatible object.

The example below shows a Prolog predicate that returns *true* if it can successfully send the message *size* to the object in the location *$0* (an instance of Person class) and the predicate stores its result in the Prolog variable *X*. In this example, the variable *X*='Ann', the name of the person that is sent as argument.

```
...
Object obj[] = {new Person};
prolog.call( "send($0,age,[],X)", obj );
...
```

3.2. Using Java objects from Prolog clauses

This connection allows the use of Java objects in a Prolog program. A Java object is like a Prolog atom, but it can receive messages. Prolog has been extended to send messages to Java objects embedded in a Prolog program. By means of these extensions it is possible to overcome the well-known Prolog's efficiency problems.

There are two ways to use Java objects in Prolog:

- Creating new instances of a class in a clause body in Prolog.
- Passing objects as arguments to the Prolog interpreter, and then using the objects in a clause body.

The creation of new instances of a class is made by the new predicate. It receives three arguments: Class, Arguments, Object; when *new(Class, Arguments, Object)* is evaluated, it generates a new instance of *Class* using the constructor with the same number and type of arguments as *Arguments*, finally it stores the new object in *Object*.

For example, the evaluation of *new('java.util.Vector',[10],Vec)* generates a new instance of *java.util.Vector* using the constructor that receives an integer (in this case, the number 10) as argument, then it stores the new vector in *Vec*.

It is also possible to send messages to Java objects from a clause body using an especial Prolog predicate: *send*. The send predicate allows the sending of a message to a Java object. The message can include arguments. It supports two types of arguments: Prolog objects or Java objects.

The evaluation of *send(Object, Message, Arguments, Result)* has the following steps:

1. It obtains the runtime class of the object *Object*.

2. It obtains the public member methods of the class of *Object* and its superclasses. After that, for each method m_i:

(a) If the name of the method m_i is *Message* and the number of arguments of m_i is equal to the length of *Arguments*, then, each element a_i of *Arguments* is converted to the same class of the *i*-th method's formal parameter type.

(b) If no method matches, the method *send* fails.

3. The method m_i is invoked with *Arguments*.

4. If m_i returns an object, it is converted to a Prolog-compatible form.

When a Java object sends a message to an object it knows the class of the object, the message's name and method's formal parameter types. These data are provided at compile-time by the Java compiler. Prolog does not have all the information about classes and methods, because the send predicate is not compiled. For this reason, JavaLog obtains the information that describes classes and methods at runtime.

There are four rules that describe the compatibility between Java and Prolog types. These rules are applied when the send predicate is evaluated, and the arguments of the message include a Prolog object:

1. If the parameter type is consistent with the formal parameter type of the message, no explicit conversion is done.

2. If the formal parameter type is consistent with *String*, the parameter is converted to a *String*.

3. If the formal parameter type is consistent with *int*, a conversion of the parameter to *Integer* is made.

4. If the parameter type is a wrapper of a Java object, compatibility between the parameter and a Java object is verified.

When JavaLog evaluates the send predicate it only knows the receptor of the message, the message name and the arguments. With this information, JavaLog obtains the object class and its superclass. Then, it searches a method with the desired name and compatible arguments. Finally, if the method is found, it is invoked.

The inclusion of Java objects in Prolog is made possible by using *wrappers*. A Java object with an associate wrapper acquires the same behavior than a Prolog atom. In this way, a Java object within Prolog is like an atom, but it can be used in the send predicate.

The next paragraphs show an example of the use of Java objects inside Prolog clauses:

> There is an intelligent agent that needs to use a planning algorithm to generate a plan to achieve its goals. The planning algorithm has been written in Prolog, using all its capabilities in unification and backtracking. The result of the algorithm is a plan, that is, a set of partially ordered actions that the agent has to follow. The plan is represented by a directed graph. The planning algorithm uses another graph to detect when a newly introduced action interferes with past decisions.
>
> In the described situation, a typical representation for a graph using Prolog is a list containing the edges. Each element of the list is a pair $[a_i,b_i]$that represents an edge (a_i,b_i) in the graph. The algorithm needs to known the existence of an edge. This

action involves a search over all the list of edges. In Java, in contrast, the same results can be achieved by using an adjacency matrix, in which an edge (a_i,b_i) appears in the matrix as an element in the position i, j. Thus, to know the existence of an edge in the graph using an adjacency graph it is only necessary to read one position of the matrix.

By using JavaLog it is possible to implement the planning algorithm in Prolog taking advantages of preconditions matching and backtracking and to use Java for implementing the action graph taking advantageous of the Java efficiency achieved in the representation and searching in graphs.

The usage of Java objects in a Prolog program requires a special treatment, since an object with an associated wrapper does not have the same behavior than standard Prolog atoms. A Prolog variable can change its state only once; on the other hand, a Java object can change its state every time that it receives a message. It affects the normal way of Prolog programs since objects changes their state during the normal recursion. The cause of this is that a Java object with a wrapper associated is only a reference to a Java object.

The existence of Java objects inside Prolog clauses has one important implication: in a recursive Prolog clause that uses Java objects the programmer has to consider the necessity to save/restore Java objects at the beginning and end of a clause respectively.

Two implementations of the POP [Weld, 1994] planning algorithm have been made to measure the improvements of JavaLog over traditional Prolog. One of the experiences has been made using only Prolog. The other experience has been implemented using JavaLog in which Prolog was used for implementing the general planning algorithm and Java was used to manage the action graph of restrictions.

These two versions of the algorithm have been tested using the Sussman anomaly problem as input. The implementations were executed using the following resources: Pentium 233 Mhz, 32 MB of RAM, JDK 1.1.3 on GNU/Linux 2.0 and JavaLog.

After ten iterations, the results show the potentiality of the integration offered by JavaLog:

- Using only Prolog: 20.124 sec.
- Using JavaLog: 4.047 sec.

The difference in performance is due to the representation of the directed graph of restrictions in Java by using an adjacency matrix. In this way, the time $O(n)$ (n is the number of restrictions) that takes the process of consistency check in the Prolog version of the algorithm is reduced to $O(1)$ by combining Java and Prolog.

4. OWB: Integrating Smalltalk and Prolog

Object With Brain (OWB) integrates Smalltalk objects and Prolog clauses allowing objects to define part of its private knowledge with logic clauses and methods implemented partially or fully in Prolog. The design of this integration is based on the following points:

1. Meta-objects which manage knowledge in logic format as a part of objects. These objects have no conscience about meta-level that adds this functionality.

2. Logic modules that encapsulate logic clauses. These modules can be located in instance variables and methods, and they can be combined for using in queries.

3. The possibility that objects can become clauses and that clauses can use objects as constant type.

In the following section, details of the integration of Smalltalk-Prolog-Smalltalk are exposed.

4.1. Smalltalk objects using Prolog

Simple objects, generally, have not the capability to manage knowledge in logic format. The possibility that old or new objects manage this type of knowledge will make feasible that these objects combine and infer knowledge without using complex algorithms. By using meta-objects, this problem has been solved. A meta-object with knowledge associated to a particular object allows the usage of a protocol defined to manage knowledge in logic format.

On the other hand, in OWB, an object may have instance variables of any object class, including objects of the *LogicModule* class. This class of objects represents logic modules defined as sets of clauses expressed in Prolog syntax. A logic module encapsulates a set of clauses and it can be combined in defined ways. The logic modules aim the modularization of logic programs.

In this way, an object can have private knowledge expressed in logic form, through rules and facts, which are available only in methods of the own object class. An object can have zero, one or more instance variables referring clauses, allowing thus the separation of concepts that the developer wishes to record in different variables. For example, let a *Professor* class that define instance variables in which each professor can register his way for evaluating students of a course, for accepting requests of new students and for altering his schedule.

OWB allows classes to use logic modules as method parts. This enables classes to record facts and rules that represent common knowledge for their instances.

The logic modules defined in class methods represent common knowledge of the objects of that class. Those logic modules that are defined in the instance variables of objects represent proper knowledge of each object. Figure 1 shows a distribution of logic modules.

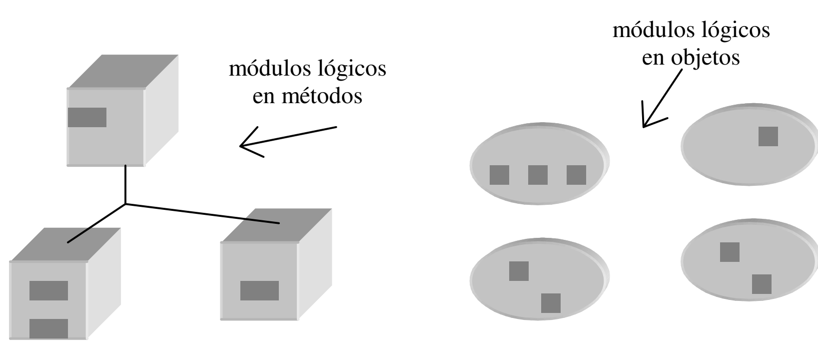

Figure 1 – Logic modules.

An important point in the use of variables with logic knowledge is that an object can have some instance variables to register different views of the same concept. These views can be used separately or can be combined using operators defined for such goal. For example, the *Professor* class above mentioned may have different instance variables (*a*, *b* and *c*) to register different ways for evaluating changes of his schedule from some request. In this way, a professor, in front of a particular situation, can use one of these forms (achieved by one of these variables) or one of its combinations.

The following operators have been defined and implemented by combining logic modules referenced by variables:

- re-write: let the knowledge bases *a* and *b*, "*a reWrite b*" define a logic module that contains all clauses defined in *b* added to the clauses defined in *a* whose head name is not the same of some clause of *b*.
- plus: let the knowledge bases *a* and *b*, "*a plus b*" define a logic module which contains all clauses of *a* and *b*.

Figure 2 shows how an object may have multiple instance variables with logic knowledge and how this object can be combined using the plus operator. The *addKnowledge()* message make available the logic module sent as argument in *knowledge meta-object* associated with the base object. This knowledge can be queried from this moment.

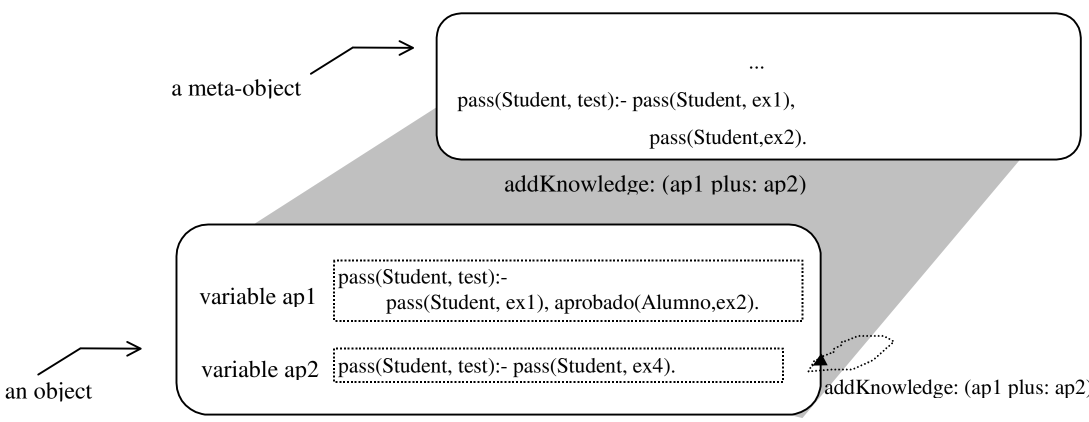

Figure 2 - Combining logic modules.

Furthermore, an object can have defined in its class methods, which are written in Smalltalk 80, both methods fully implemented in logic and methods that combine Smalltalk and Prolog.

This integration allows the combination of Smalltalk and Prolog syntax in a method to express declarative knowledge in declarative form and operational behavior in procedural form. However, both forms of programming share the same world. For this reason, both forms can access to the same information. So, objects can work with clauses and clauses can work with objects.

4.2. Prolog using Smalltalk objects

In the body of Prolog clauses it is possible to send messages to objects, to create new objects and to use objects as atoms.

Furthermore, a logic module in a method, which is between double braces, can use local, global, or class variables and any method arguments directly in its clauses. The following example shows how the student referenced by *anStudent* variable, which is passed as parameter of *eval* method is used in *qualification* clauses.

```
eval: anStudent
{{qualification({anStudent}, 'A') :-
                finalTest({anStudent, passed).
  qualification({anStudent}, 'B') :-
                finalTest({anStudent}, unpassed),
                exercises({anStudent}, passed).}}
```

5. Conclusions

In this paper the basis for the development of software intelligent agents from the programming point of view has been presented. Two alternatives were presented. The difference between the presented options is based on the typed characteristics of programming languages used. Smalltalk allows the easy usage of dynamic structures such meta-objects. Java in contrast involves code preprocessing and the necessity to consider types compatibility.

On the other hand, the fact of that the Prolog interpreter was implemented in the proper language allows extensions to this interpreter. These extensions can supports the management of mental attitudes.

6. References

[Amaral, 1993] J. Amaral, Um Estudo sobre Comportamento Inteligente, Technical Report CPGCC of UFRGS, Porto Alegre, Jan. 1993.

[Ciancarini, 1997] P. Ciancarini, A. Knoche, D. Rossi, R. Tolksdorf, F. Vitali, Coordinating Java Agents for Financial Applications on the WWW, Proc. of The Practical Application of Agents and Multi-Agents Technology, Apr. 1997, pp.179-191.

[Cohen, 1990] P. R. Cohen, H. J. Levesque, Intention is Choice with Commitment, Artificial Intelligence, Vol. 42, No. 2, 1990.

[Fukunaga, 1986] K. Fukunaga, S. Hirose, An Experience with a Prolog-Based Object-Oriented Language, Sigplan Notices (Proc. of OOPSLA '86 Conference), Nov. 1986, pp. 224-231.

[Fisher, 1994] M. Fisher, Representing and Executing Agent-Based Systems, ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Aug. 1994, pp. 307-323.

[Huang, 1996] Z. Huang, M. Masuch, L. Pólos, Alx, an Action Logic for Agents with Bounded Rationality, Artificial Intelligence, Vol. 82, No. 1, 1996, pp. 75-127.

[Ishikawa, 1986] Y. Ishikawa, M. Tokoro, A Concurrent Object-Oriented Knowledge Representation Language Oriente84/k: It's features and implementation, SIGPLAN Notices, Vol. 21, No. 11, Nov. 1986, pp. 232-241.

[Kolb, 1995] M. Kolb, A Cooperation Language, Proc. of the International Conference of Multi-agent Systems, 1995, pp. 233-238.

[Mello, 1987] P. Mello and A. Natali, Objects as Communicating Prolog Units. Proc. of ECOOP'87 European Conference on Object-Oriented Programming, Jun. 1987, pp. 181-191.

[Marcarella, 1995] P. Marcarella, A. Raffaetà, and F. Turini, Loo: An Object-Oriented Logic Programming Language, Proc. of Italian Conference on Logic Programming (GULP '95), Sep. 1995.

[Poggy, 1994] A. Poggy, Daisy: an Object-Oriented System for Distributed Artificial Intelligence, ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Aug. 1994, pp. 341-354.

[Rao, 1991] A. Rao, M. Georgeff, Modeling rational agents within a BDI-architecture, Proc. of knowledge, representation and reasoning (KR '91), pages 473-484, April 1991.

[Shoham, 1993] Y. Shohan, Agent-Oriented Programming. Artificial Intelligence, Vol. 60, No. 1, Mar. 1993, pp. 51-92.

[Vaucher, 1988] J. Vaucher, G. Lapalme, and J. Malenfant, Scoop: Structured Concurrent Object-Oriented Prolog. In Proc. of ECOOP'88 European Conference on Object-Oriented Programming, pages 191-211. Springer-Verlag, August 1988.

[Linder, 1996] B. van Linder, W. van der Hoek, and J.-J. Ch. Meyer, Formalising Motivational Attitudes of Agents: On preferences, goals and commitments. In M. Wooldrige, J. Muller, and M. Tambe, editors, Intelligent agents II, pp. 17-31. Springer, New York, 1996.

[Weld, 1994] Daniel S. Weld, An Introduction to Least commitment Planning, AI Magazine, Vol. 15, No. 4, 1994, pp.27-61.

[Weerasooriya, 1994] D. Weerasooriya, A. Rao, K. Ramamohanarao, Design of a Concurrent Agent-Oriented Language, ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Aug. 1994, pp. 386-401.