Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Computer Languages, Systems & Structures 31 (2005) 17–33

COMPUTER LANGUAGES, SYSTEMS & STRUCTURES

www.elsevier.com/locate/cl

JavaLog: a framework-based integration of Java and Prolog for agent-oriented programming

Analía Amandi, Marcelo Campo, Alejandro Zunino*

ISISTAN Research Institute. UNICEN University, Campus Universitario, Paraje Arroyo Seco, Tandil (7000), Buenos Aires, Argentina

Received 6 November 2003; accepted 25 March 2004

Abstract

Intelligent agent development has imposed new challenges on the necessary language support. Object-oriented languages have been proposed as an appropriate tool, although logic-oriented languages are more adequate for managing mental attitudes. Multi-paradigm languages supporting encapsulation of actions, hiding of private knowledge and flexible manipulation of knowledge are, certainly, a good alternative for programming agents. However, a unique language to support flexible and efficient development of multi-agent systems confronts with the tradeoffs imposed by expressive power, efficiency and support technology. An alternative to conciliate these tradeoffs is not to think about a single language but an incrementally compatible family of agent-oriented multi-paradigm languages. In this work we present an approach based on object-oriented framework technology for integrating object and logic paradigms in such a way that new language features can be incrementally added to the core language. This core language is based on logic modules integrated as object abstractions in the object paradigm. JavaLog is a materialization of this framework integrating Java and Prolog. This core was extended to provide multi-threading support, mobility and temporal-logic operators to Prolog. MoviLog, the mobile part of the family provides a novel mobility mechanism, reactive mobility by failure, which enables virtual Prolog databases distributed across Web sites.
© 2004 Elsevier Ltd. All rights reserved.

Keywords: Multi-paradigm languages; Logic programming; Object-oriented programming; Agent-oriented programming; Object-oriented frameworks; Intelligent agents

* Corresponding author. Tel.: +54-2293-440363; fax: +54-2293-440322.
E-mail addresses: amandi@exa.unicen.edu.ar (A. Amandi), mcampo@exa.unicen.edu.ar (M. Campo), azunino@exa.unicen.edu.ar (A. Zunino).

1477-8424/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2004.03.001

1. Introduction

Intelligent agent programming requires particular capabilities of a programming language. This fact relies on both experiences gained developing applications with standard languages and from the many proposed agent-oriented languages [1].

Many applications have been developed using object-oriented languages and characteristics of these languages such as encapsulation and inheritance are considered important benefits for agent development [2,3]. On the other side, several proposals for agent languages such as Metatem [4] or Gaea [5], among others, are supported on logic bases. A logic-oriented programming approach is a straightforward consequence of the requirement of managing mental attitudes, since they are generally based on special logics defined for that purpose [6]. For this reason, logic languages such as Prolog may be considered an obvious way for representing and inferring relationships among mental attitudes such as intentions, goals and beliefs. However, they present several limitations for the definition of action capabilities that object-oriented languages naturally support. Usually, logic-based agent languages are enhanced with non-declarative features inherent to other type of languages, such as objects, threads, etc. Thus, in order to have a language that makes it easier the development of different kind of agents, this language should provide, at least, support to smoothly manage mental attitudes, actions, and private information in an integrated way.

Certainly, multi-paradigm languages [7–11] integrating logic and object-oriented paradigms represent a convenient choice for the definition of agent programming languages. However, the existing proposals of multi-paradigm languages, particularly the ones proposed in the 1980s and their evolutions, present limitations to satisfy the multiple requirements of modern multi-agent systems. In addition, flexibility to cope with specific language capabilities required by the many potential application areas is an essential factor. Also, as efficiency is involved, the use of a language providing a set features not necessary for the domain can be negative, in terms of memory usage or speed, despite the expressive power of the language. Therefore, the main challenge behind the design of an effective language for agent programming relies on how to maintain the right balance among expressive power, support technologies and functionality. We believe that object-oriented framework technology [12,13] is the current most viable answer.

Under these considerations we have designed JavaLog, a language that integrates Java and Prolog, enabling to exploit the advantages of both programming paradigms in an extensible way. JavaLog is based on a Java framework that can be extended to incrementally add new functionalities to the core language, giving a family of languages. The core language is based on logic modules that encapsulate clauses for the manipulation of mental attitudes. These modules are used through variables or directly in Java methods, defining precise bridges between the two paradigms. Features such as temporal Prolog extensions, concurrency or strong mobility, optional for some agent systems, are provided as incremental extensions of the framework. Concurrency is supported by Java threads; logic threads are also supported. Strong mobility is incorporated supporting logic modules that can be transparently executed using clauses at different sites.

In the remainder of the article, we will first describe our approach based on logic modules for agent programming, then, in Section 3 we summarize relevant characteristics of the JavaLog language. The JavaLog framework is briefly described in Section 4. Then, Section 5 describes an extension of the JavaLog framework to support concurrency. Mobility extensions and a novel form of mobility are presented in Section 6. In Section 7 we describe a number of applications of the framework. Section 8 discusses the most relevant related work. Finally, in Section 9 we conclude with a summary and directions for future work.

2. Integrating logic and object paradigms

The object-oriented and logic paradigms work over different conceptual worlds. The integration of these worlds presents several problems, which are a consequence of the different nature of the computational elements involved and the way in which they are manipulated in each paradigm.

The object-oriented paradigm works over objects, which are only accessed by methods. The logic programming paradigm is based on logic clauses, which manipulate a defined set of terms.

Our approach for agent-oriented programming defines logic modules as the basic components for paradigm integration. Java provides modularization from the object-oriented paradigm and, in this context, logic modules are incorporated as modules encapsulating Prolog clauses. This integration allows uniformity in both the definition and manipulation of agent mental states.

Logic modules are defined as a set of Horn clauses following the definition of O'Keefe [14]. This gives two algebraic operators, union and overriding union [15] for combining logic modules.

Java classes can define part of their methods using logic modules and objects. Classes can also define private modules in instance variables. A simple object associated with another object that we call *brain* composes an agent. This brain of the object is an instance for the logic interpreter responsible for the logic module management. This logic interpreter can be an instance of our Prolog engine, any of the extensions that we provide, or one of the extensions that a developer can add for particular proposes.

JavaLog allows us to define logic modules into Java variables or within Java methods. To clarify the presentation, we will introduce an example, a salesman agent. It has the ability to select and buy items based on user preferences. *CommerceAgent* is a class defined for implementing this type of agent.

In order to allow flexibility on the article selection, a logic module defines preferences. Below, a logic module expresses the preferences of a user in buying a vehicle. Here, *send* is used for sending a message to a Java object from a Prolog clause. For instance, $send(vehicle,type,[],T)$ in Prolog is equivalent to $t = vehicle.type()$ in Java.

```
preference(vehicle, 10) :-
  send(vehicle,type,[],T), T = car,
  send(vehicle,model,[],M), M > 2000,
  send(vehicle,price,[],P), P > 20000.
preference(vehicle, 9) :-
  send(vehicle,type,[],T), T = car,
  send(vehicle,model,[],M), M = 2001,
  send(vehicle,size,[],S), S = big.
```

Now, we expose the *CommerceAgent* class written in JavaLog:

```
public class CommerceAgent {
  private PlLogicModule userPreferences;
  public CommerceAgent( PlLogicModule userPrefs ) {
    this.userPreferences = userPrefs;
}
....
public boolean buyArticle( Article art1,art2 ) {
  userPreferences.enable();
  ?-preference(#art1#,Pref1).;
  ?-preference(#art2#,Pref2).;
  if (Pref1 > Pref2) {
    buy(art1)
  } else {
    buy(art2)
  };
  userPreferences.disable();
}
```

We first define a variable *userPreferences*, which contains a logic module representing user preferences. Then we expose a method for helping to decide which of the two articles to buy, considering the user preferences. Thus, the *buyArticle* method first places the *userPreferences* logic module as available for the Prolog interpreter (the agent *brain*). Then, two Prolog queries are used to determine the level of preference of each article. Each Prolog query is introduced by "?-", following the usual Prolog syntax. To evaluate *preference*(#*art*1#, *Pref* 1), the clauses in *userPreferences* are used, since they were explicitly enabled. Each query refers to the name of a Java variable enclosed by "#". This allows us to use existing Java objects inside a Prolog clause. Finally, the method ends disabling the *userPreferences* logic module. This operation removes the logic module with the user preferences from the agent brain.

We have shown a big picture of the ideas upon which JavaLog is based. The next sections detail how the interaction between the paradigms is accomplished and what are the advantages that JavaLog provides for agent-oriented programming.

3. Integration schemes

Several interaction alternatives among objects and logic modules are defined in JavaLog. We call these alternatives as *interaction schemes*. Interaction schemes are classified as integration by reference, value and composition schemes. Reference specify the composition limits of different modules. Value specify the role of objects in logic modules and the role of logic variables in methods. Composition specifies how logic modules can be combined, expressing also the composition of a knowledge base when a query is executed.

3.1. Integration by reference

Integration by Reference means that logic modules can be located in instance variables and as part of methods. Thus, an agent can be designed as an object with private logic knowledge associated with him. The action capabilities of the agent are represented by methods enabled for using logic knowledge.

Considering the previous example, we can define another variable in the agent with different preferences that have to be applied when the user is experiencing bad financial times. In this logic module, we can define that the user prefers to buy economy cars.

The fact that an agent records different logic modules in variables does not imply that the agent uses all these modules in the logic queries. The agent will only use the logic modules made available to his brain. Thus, several combinations can be made when he is reasoning.

The definition of logic modules within Java methods allows agent developers to specify mental attitudes that are common to all the agents of such a class. Thus, for example, common preferences on cars can be specified directly in a method instead of using variables.

Variables used in logic queries inside methods can also be used in the Java code of the rest of the method.

Objects do not naturally have the capability of managing clauses. By using a logic language interpreter integrated with our Java objects, this obstacle for programming agents has a solution. An instance of a logic language interpreter associated with a Java object allows the definition of object-agents. These object-agents are composed of an object that can manage actions and communication messages and a logic interpreter that manages mental attitudes in a logic format.

In this sense, an object-agent can have private mental attitudes expressed in logic form, by means of rules and facts, manipulated through methods of the object-agent class. An object-agent can have zero, one or more instance variables referring to logic modules, allowing the separation of concerns that the agent wishes to apply in different contexts.

Our approach allows classes to define logic modules in methods. This enables classes to record facts and rules that represent common attitudes for their instances. The logic modules defined in methods represent common knowledge of the objects of such a class. Those logic modules defined in the instance variables of objects represent proper mental attitudes of each object.

3.2. Integration by value

Integration by value specifies communication bridges between both object-oriented and logic paradigms. Thus objects can become Prolog facts and be part of a logic module. Also, clauses can use objects as a special kind of term. Finally, logic terms used in clauses in a method can be manipulated in the Java parts of the method and vice versa.

Single objects in JavaLog can become a fact receiving a message named *asClause*. For instance, if an object of *Person* class with three instance variables with values *Ann*, 36 and *engineer* receives that message the following clause is generated: *person*(*herself*, ‘*Ann*’, 36, *engineer*). This fact is named as the receiver object class. The first argument is an object (indeed, a person object, the receiver of the method), the other arguments are the contents of the instance variables of the receiver object (name, age and profession in this case).

Following the second rule, clauses can use objects as a kind of term. This allows the direct usage of objects together with their associated behavior (the methods defined in their class and super classes) inside logic code. For sending a message to objects from a logic code, a predefined clause named send is used. Thus, *send*(*aPerson*, *age*, [], A) into the body of a clause produces the sending of the message *age* to the object *aPerson* without any argument, instantiating the variable A with the number that returns that message.

Following the third rule, logic terms used in clauses in a method can be manipulated in all the non-logic parts of the method and vice versa. Instance variables, class variables, and local variables are available in methods. Logic variables are available in clauses. In JavaLog, any Java variable available in a method can be used in any clause that is part of a logic module inner to that method. Also, any variable used in a clause can be used in the method it is part of.

3.3. Integration by composition

Composition rules limit the possibilities of combining logic modules. Algebraic operators for logic modules [15] have been applied from two different perspectives. The first combination perspective establishes that modules referred by variables can be directly combined using predefined methods. From the second perspective, logic modules as part of a method can be combined by inheritance.

An important point about the usage of variables referring to logic modules is that an object-agent can have different instance variables to register different views of the same mental attitude. These views can be used separately or can be combined using operators defined for such goal. For example, considering a *PersonalAssistant* class, it may have different instance variables (a, b and c) to register different ways for evaluating changes of its schedule from some request. In this way, an assistant agent, in front of a particular situation, can use one of these forms (achieved by one of these variables) or some of its combinations.

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

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

The operator *re-write* follows the algebraic definition of overriding union and the operator *plus* follows the definition of union.

On the other hand, logic modules can be defined into methods. These logic modules can also be combined using the same operators. Fig. 1 shows how these combinations can be done. Each column shows each kind of combination. Both columns expose the method *method*( ) into a class A and one subclass B.

The first column shows the application of the union operator. Each method defines different rules for preferences. In this case, an object, instance of class A, uses the preference defined between {{ and }} in its queries after the invocation of that method. An object, instance of class B, has

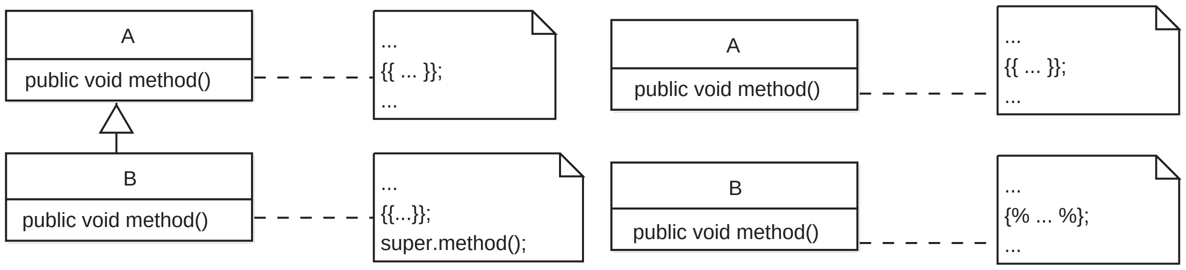

Fig. 1. Operations with logic modules.

two options in terms of the definitions of preferences to be used. The developer has the option of explicitly invoking *method*() of the superclass, using in this case the addition of both definitions. The place of the invocation defines the ordering of the clauses, thus it is important whether the invocation is before or after the definition of the local logic module. The second option is not to invoke the method of the superclass, resulting in a complete redefinition of the method.

The second column shows the application of the operator overriding union. Here each method also defines different rules for preferences. An object *A* has the same behavior as that exposed in the first column, but an object *B* has the option of combining the logic modules defined in *method*() of the superclass and its own logic modules. The marks {% and %} indicate the modules of the superclass must be *re-written* by the clauses between the special marks.

4. The framework

JavaLog is designed as an object-oriented framework [12,13] to add new language features as needed. Fig. 2 shows a class diagram with the most important classes of the framework. The fundamental concept on which the language is based is the logic module, which is represented by the class *LogicModule*. A *LogicModule* is a sequence of Prolog clauses (class *PlClause*). These fundamental constructs are extended to support extra features such as modules with temporal operators (class *TemporalLogicModule*), clauses containing Java objects (*PlJavaObj*), etc. In a similar way, the core JavaLog interpreter (class *BasicLogicInterpreter*) can be extended to support new language capabilities such as concurrency (class *MultiTreadedBraind*) strong mobility (class *MoviLogBrainEngine*) and integration with Web servers (class *MARlet*).

As an example of extensibility we will describe how to support a special type of mobile agents named *Brainlets* [16]. First, we extend *MultiThreadedBrain* defining a class *Brainlet*. For supporting strong mobility of Brainlets, we have to be able to save/restore its execution state, suspend/resume an executing Brainlet and take into account hops between sites when backtracking. In order to add these capabilities to the language we add two methods *getBrainState* and *setBrainState*. The first one returns the execution state of a Brainlet as a *BrainState* object containing the execution stack, program counter and variables of the Brainlet. The second method is used to set the internal state of a new Brainlet. To stop/resume the execution of a Brainlet we simply add an instance variable *executing* to the class that is *true* when the Brainlet is executing and *false* otherwise. This variable is modified by *suspendExecution* and *resumeExecution*. To handle the variable *executing* we just

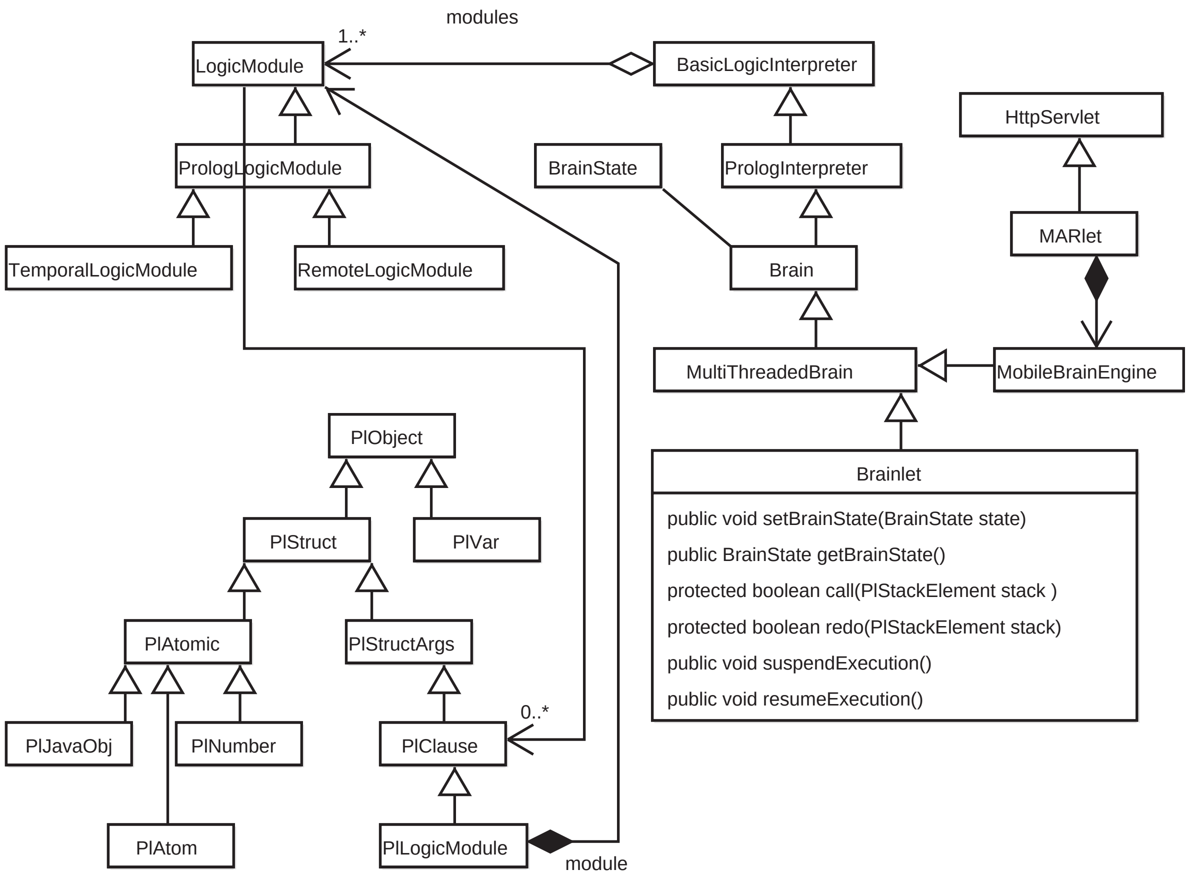

Fig. 2. Partial JavaLog class hierarchy.

override the methods *call* and *redo* which implement the depth first execution strategy of Prolog. In these two methods we also add logic to handle distributed backtracking.

When a Brainlet wishes to migrate to a remote site it invokes its *move* method which executes the following sentences:

```
suspendExecution();
BrainState state=getBrainState();
AgentManager.sendState(host, state);
```

At the remote site the following code is executed to resume the execution of the Brainlet:

```
Brainlet b = new Brainlet();
b.setBrainState(receivedState);
b.resumeExecution();
```

As shown above, the JavaLog framework can be extended with little effort. In opposition, most WAM-based Prolog implementations are difficult to extend due to their architectural design.

5. Thinking concurrently

Complex cognitive agents may need to think concurrently about their goals, choices and future courses of action. For this, JavaLog was extended to provide a mechanism to create and synchronize threads. We call this extension JavaLog©.

The clause *thread*(G_1), $G_2, \ldots, G_n$ where G_i are goals, creates a new JavaLog© thread which tries to prove G_1 in a new thread, while the execution of $G_2, \ldots, G_n$ continues. For example, *thread*(*search*(*Pr*, *A*)), searches a list of articles matching a number of preferences. If it succeeds *A* it is instantiated with a list of articles that match *Pr*.

The *wait*(*Variable*) synchronization predicate allows the synchronized communication of two threads by using JavaLog© variables. For example, an agent may concurrently search for three articles matching different preferences:

```
thread(search(Pr1, A)),thread(search(Pr2, B)),
thread(search(Pr3, C)),..., wait(A), wait(B), wait(C).
```

In order to obtain the results of the three searches, the agent has to wait for the instantiation of *A*, *B* and *C*. The *wait* clause suspends the execution of the current thread until the associated variable is instantiated by another thread.

In addition to these two mechanisms, JavaLog© handles concurrency at the level of logic modules and provides facilities to prevent conflicts among threads. For example, a thread may remove a clause that is needed by other threads, thus causing their failure. These conflicts can be handled by a locking mechanism that prevents the removal of specific clauses. For example, *lock*(*append*/2) disallows the removal of all the clauses *append* with two arguments, while *unlock*(*append*/2) performs the inverse.

6. Mobility

The huge amount of information available on the Internet became one of the main motivations for the development of mobile agent technology [17,18]. Such a capability is particularly interesting when an agent makes sporadic use of a valuable shared resource. But also, efficiency can be improved by moving agents to a host to query a large database, as well as, response time and availability would improve when performing interactions over network links subject to long delays or interruptions of service. For this, a further extension of JavaLog© was developed, called MoviLog [16].

MoviLog is, essentially, an extension of JavaLog© to support mobile agents across Web servers. MoviLog implements a strong mobility model for a special type of logic modules, called Brainlets. The MoviLog inference engine is able to, besides processing several concurrent threads, restart the execution of an incoming Brainlet at the point where it migrated.

In order to enable mobility across Web sites, each Web server belonging to a MoviLog network must be extended with MARlets (Mobile Agent Resources). A MARlet extends the Java servlets support encapsulating the MoviLog inference engine and providing services to access it. In this way, a MARlet represents a Web dock for Brainlets. Additionally, a MARlet is able to provide intelligent services under request, such as adding and deleting logic modules, activating and deactivating logic modules, and answering logic queries. In this sense, a MARlet can also be used to provide inferential services to Web applications or agents.

From the mobility point of view, MoviLog provides the support to implement Brainlets with typical pro-active capabilities, but more interesting yet, it implements a mechanism for transparent reactive mobility by failure.

6.1. Proactive mobility

MoviLog adds the *move_to* built-in predicate, which allows a Brainlet to autonomously migrate to another host. Before transport, the MoviLog engine in the local host serializes the Brainlet and its state—i.e. its knowledge base and code, current goal to satisfy, instantiated variables, backtracking points, etc. Then, it sends the serialized form to its counterpart on the destination host. Upon receipt of an agent, the MoviLog machine in the remote host reconstructs the Brainlet and the objects it refers to, and then it resumes its execution. Eventually, after performing some computation, the Brainlet could return to the originating host calling the return built-in predicate.

The following example presents a simple Brainlet for e-commerce, which has the goal of finding and buying a given article in the network according to the user's preferences.

```
Brainlet CustomerBrainlet = {
  sites([www.offers.com,www.freemarket.com,...]).
  preference(car,[ford, Model, Price]) :-
    Model > 1998, Price < 60000.
  preference(tv,[sony, Model, Price]):-
    Model = 21in, Price < 1500.
lookForOffers(A,[],_,[]).
lookForOffers(A,[S| R], [O|RO], [O| Roff]):-
  move_to(S), article( A, Offer, Email),
  O= (S,Offer,Email), lookForOffers(A, R, RO,ROff).
lookForOffers(A,[S| R], [O|RO], [O| Roff]):-
  lookForOffers(A, R, RO,ROff).
buy(Art):-
    sites(Sites),
    lookForOffers(Art, Sites,R,Offers),
    selectBest(Offers, (S,O,E)), move_to(S),
    buy_article(O,E), return.
  ?- buy(#Art).
}
```

The *buy* clause look for offers available in the different sites, select the best and calls a generic predicate to buy the article (this process is not relevant here). The *lookForOffers* predicate implements the process of moving around the defined sites looking for the available offers for the article (we assume that we get the first offer). If there is no offer in the current site, the Brainlet goes to the next one in the list.

Although proactive mobility provides a powerful tool to take advantage of network resources, in the case of Prolog, it also adds an extra complexity due to its procedural nature. That is, mobile Prolog programs cannot necessarily be built in the declarative way as a normal Prolog program is, forcing the implementation of solutions that depend on the mobility aspect. Particularly, when the mobile behavior depends on the failure or not of a given predicate solutions tend to be more complicated. This fact led us to develop a complementary mobility mechanism, called reactive mobility by failure.

6.2. Reactive mobility by failure

Reactive Mobility by Failure (RMF) is a novel mobility mechanism which aims at reducing the effort of developing mobile agents by automating some decisions about mobility [16]. RMF is based on the assumption that mobility is orthogonal to the rest of attributes that an agent may possess (intelligence, agency, etc.) [19]. Under this assumption it is possible to think of a separation between these two functionalities or concerns at the implementation level [20]. RMF exploits this separation by allowing the programmer to focus his efforts on the stationary functionality, and delegating mobility issues on a distributed multi-agent system that is part of the MoviLog platform.

Agents that use the underlying RMF mechanism are called *Brainlets*. A Brainlet is composed by Prolog clauses, Java objects and a number of *protocols*. A protocol is a declaration of the interface used to access a resource across the network. By resource we mean, for example, data (Prolog clauses, databases, Web pages, Java objects, etc.) or code (Prolog clauses, Web accessible programs, Java methods, etc.).

When a predicate of a Brainlet declared as a protocol fails (it is not possible to evaluate that predicate at the current site, or there are no more choices left), RMF transparently moves the Brainlet to another site having definitions for such a predicate and continues the normal execution to try to find a solution (Fig. 3). The implementation of this mechanism requires the MoviLog inference engine, named MARlet, to know where to send the Brainlet which implies that the MARlet has to know the protocols available at every site of the network. In order to maintain this information, some of the mobility agents named PNS (protocol name servers) discover MARlets and keep up to date the protocols offered by each site.

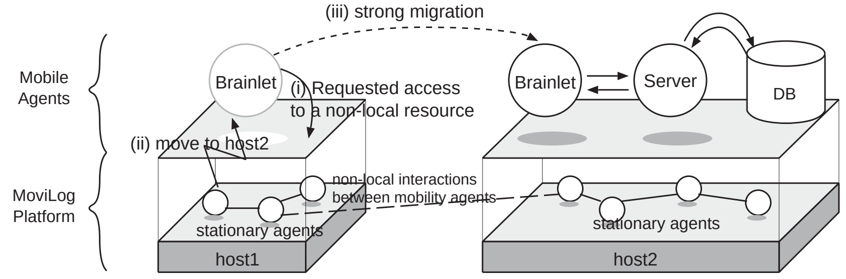

Fig. 3. Reactive mobility by failure.

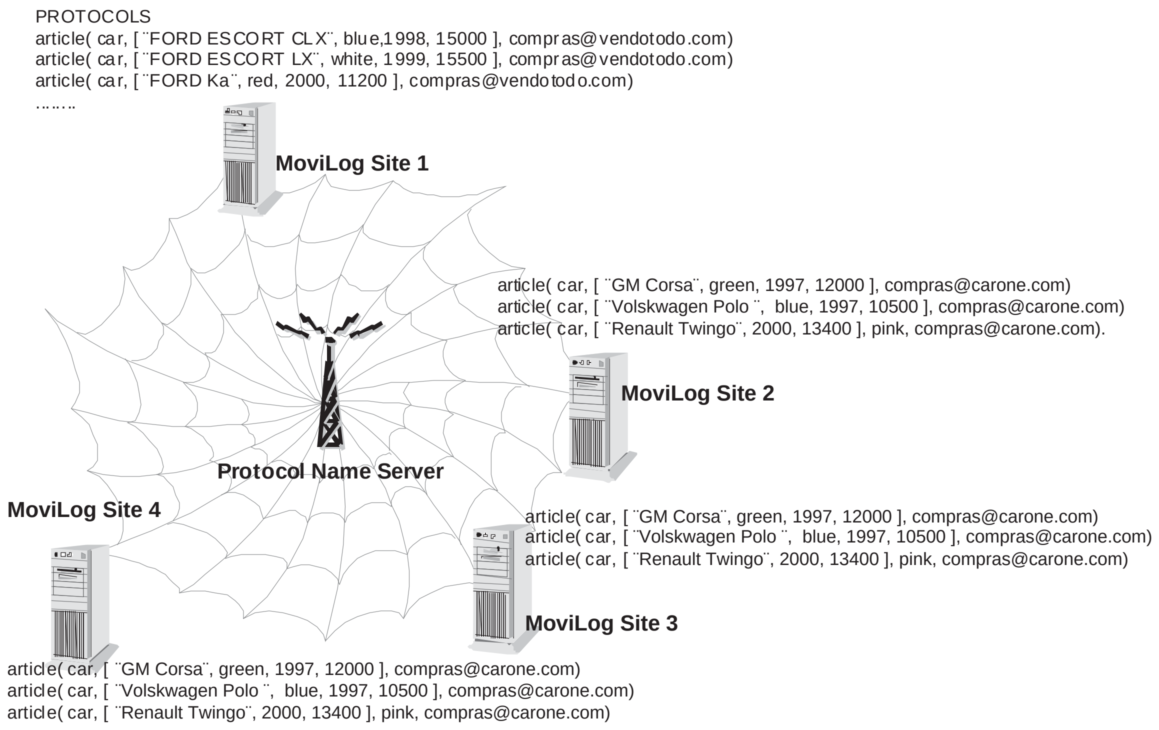
PROTOCOLS
article(car, [¨FORD ESCORT CLX¨, blue,1998, 15000], compras@vendotodo.com)
article(car, [¨FORD ESCORT LX¨, white, 1999, 15500], compras@vendotodo.com)
article(car, [¨FORD Ka¨, red, 2000, 11200], compras@vendotodo.com)
....
MoviLog Site 1
article(car, [¨GM Corsa¨, green, 1997, 12000], compras@carone.com)
article(car, [¨Volskwagen Polo ¨, blue, 1997, 10500], compras@carone.com)
article(car, [¨Renault Twingo¨, 2000, 13400], pink, compras@carone.com).
MoviLog Site 2
Protocol Name Server
MoviLog Site 4
article(car, [¨GM Corsa¨, green, 1997, 12000], compras@carone.com)
article(car, [¨Volskwagen Polo ¨, blue, 1997, 10500], compras@carone.com)
article(car, [¨Renault Twingo¨, 2000, 13400], pink, compras@carone.com)
MoviLog Site 3
article(car, [¨GM Corsa¨, green, 1997, 12000], compras@carone.com)
article(car, [¨Volskwagen Polo ¨, blue, 1997, 10500], compras@carone.com)
article(car, [¨Renault Twingo¨, 2000, 13400], pink, compras@carone.com)

Fig. 4. MoviLog virtual database.

In addition, MoviLog extends the normal definition of a logic module with protocol sections, which define predicates that can be shared across the network:

```
Brainlet CustomerBrainlet = {
  PROTOCOLS
    article( A, Offer, Email).
...
}
```

Protocol definitions create the notion of a virtual database distributed among several Web sites as shown in Fig. 4. When a Brainlet defines a given protocol predicate in a host h, the MoviLog engine informs the PNS, which in turn inform the rest of registered servers that the new protocol is available in h. In this way, the database of a Brainlet can be defined as a set $D = \{D_L, D_R\}$, where D_L is the local database and D_R is a list of sites that offer the same protocol clause as the current goal g. Now, in order to probe g the interpreter has to try all the clauses $c \in D_L$ such that the head of c unifies with g. If none of those lead to probe g, then it is necessary to try to probe g from one of the non-local clauses in D_R. To achieve this, MoviLog transfer the running Brainlet to one of the hosts in D_R by using the same mechanism used for implementing proactive mobility. Once at the remote site, the execution continues to try the new backtracking point. However, if the interpreter at the remote site fails to probe g, it continues with the next host in D_R. When no more possibilities are available, the Brainlet is moved to its origin.

The following example shows the implementation of the previous one combining both mobility mechanisms:

```
Brainlet CustomerBrainlet = {
  PROTOCOLS
    article( A, Offer, Email).
CLAUSES
    preference(car,[ford, Model, Price]) :-
      Model > 1998, Price < 60000.
    preference(tv,[sony, Model, Price]) :-
      Model = 21in, Price < 1500.
    lookForOffers(A, [O|RO], [O| Roff]):-
      article( A, Offer, Email),
      assert(offer((thisSite,Offer,Email)),
      fail.
    lookForOffers(A,_,Offers):-!,
      findall(offer(S,O,E), Offers).
    buy(Art):-
      lookForOffers(Art,R,Offers),
      selectBest(Offers, (S,O,E)),
      move_to(S),
      buy_article(O,E),
      return.
    ?- buy(#Art).
}
```

As can be noted, the solution using RMF looks much like a common Prolog program. This solution collects, through backtracking, the matching articles from the database until no more articles are left. The article protocol causes the Brainlet to try all the sites offering the same protocol before returning to the origin site to collect (*findall*) all the offers in the local database of the Brainlet. Once the best offer is selected the Brainlet proactively moves to the site offering that article to buy it. Certainly, this solution is simpler than the one using just proactive mobility.

6.2.1. Backtracking and consistency issues

Mobile Prolog, and particularly, the RMF mobility model generates several tradeoffs related to standard Prolog execution semantics. Backtracking is one of them. When a Brainlet moves around several places, many backtracking points can be left untried in such places, and the question is how the backtracking mechanism should proceed. The solution adopted by MoviLog resides in the PNS. The PNS provides a sequential view of the multiple databases and is used by the routing mechanism to go through the distributed execution tree.

Also the evaluation of MoviLog code in a distributed manner may lead to inconsistencies. For example, MARlets can enter or leave the system, may alter their protocol clauses or modify their databases. At this moment, MoviLog defines a policy that consists on updating the local view of a Brainlet when it arrives to a host. This involves querying the PNS to obtain a list of hosts implementing a given protocol clause and querying the local MARlet in order to obtain a list of clauses matching the protocol clause being evaluated.

7. Experimental results

JavaLog has been used in several agent-related research projects. For example, NewsAgent [21] is an intelligent agent that has the capability of generating personal newspapers from particular users' preferences extracted by observing users' behavior. This agent uses static word classification and case-based reasoning for dynamic sub-classification of interesting documents.

The QueryGuesser Agent [22] is an interface agent that assists users who work with database systems. QueryGuesser has the capability of managing personalized queries in a relational database system, according to the interests and habits of a given user.

Obelix [1] is an intelligent meeting scheduler that is able to autonomously organize users' activities taking into account their preferences, conflict among groups, travel distances, etc.

Brainstorm/J [23] is an object-oriented framework written in JavaLog which provides generic abstractions for building multi-agent systems. These abstractions defines typical agent behavior, such as perception, reaction, interaction, mobility, deliberation, etc. The main difference between Brainstorm/J and tools such as AgentBuilder [24], BDIM [25] and ZEUS [26], to name a few, is its ability to be tailored to build specific applications and extended with new functionality by reusing the abstractions, thus considerably reducing the development effort. In addition, Brainstorm/J provides mechanisms to transparently add agent abilities to any Java application.

8. Related work

Several languages have been proposed for programming agents [4,7,27–30]. Some of them use some object-oriented concepts in a logic context. For example, Concurrent Metatem [4,28] uses a set of rules based on temporal logic to represent an object's internal definition. Daisy [7] takes an object-oriented approach for building agents that communicate and derive future actions based on the speech act theory.

From the object-oriented perspective, both language extensions and frameworks for agent development have been proposed. Some of the proposed languages (i.e., [27,31]) do not take into consideration the logic fundamentals of mental attitudes. This lack could be solved using some practical views of architectures on mental attitudes, but the effort of materializing those practical approaches would have a high cost. On the other hand, agent frameworks such as [32–34], provide architectural options to build agent systems, but support few *agent* concepts thus failing when *intelligent* or *rational* behavior is required.

Our approach tries to take advantage of both kinds of proposals, defining an integration of both paradigms, and being flexible in order to accommodate different requirements.

[1] http://www.exa.unicen.edu.ar/~azunino/obelix.html

9. Conclusions

In this article, an approach for the development of intelligent agents from the programming point of view has been presented. This approach is based on the fact that the object-oriented paradigm is a good paradigm for programming agents but it presents some problems in the manipulation of mental attitudes. The problem of the manipulation of mental attitudes, usually treated on specific logic formalisms, is solved by the use of logic programming. In short, we present a multi-paradigm approach for programming agents.

This particular approach and its materialization on the JavaLog language allows flexibility for many design components. One of them is based on the fact that our Prolog interpreter has been implemented in the Java language thus allowing extensions to this interpreter. These extensions can support, for example, some particular management of mental attitudes and time.

JavaLog has been used in several developments showing cost advantages. For example, the implementation of an agent that applies a least commitment planning algorithm [35]. The main part of the algorithm was developed using Prolog, but an object modeled the graph of causal links. This implementation was contrasted with an initial implementation in Prolog. This analysis, considering these two implementations, shows that the planning algorithm using the integration runs 10 times faster than the implementation in pure Prolog [36]. The difference is based on the analysis of the graph, which in the multi-paradigm implementation uses Java.

We are developing new agents based on JavaLog. For example, we are testing an agent for analyzing electronic newspaper pages [21] and building 3D visualizations of software requirements specified in Z [37].

References

[1] Dix J, Leite JA, Satoh K, editors. Computational logic in multi-agent systems, Datalogiske skrifter, vol. 93. Roskilde, Denmark, August 1, 2002.

[2] Crnogorac L, Rao AS, Ramamohanarao K. Analysis of inheritance mechanisms in agent-oriented programming. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence, IJCAI. Los Altas, CA: Morgan Kaufmann Publishers, 1997. p. 23–29, 647–54.

[3] Shoham Y. An overview of agent-oriented programming. In: Software agents, Menlo Park, USA, AAAI; 1997.

[4] Fisher M. A survey of concurrent METATEM—the language and its applications. In: Gabbay DM, Ohlbach HJ, editors. Temporal logic—Proceedings of the First International Conference, Lecture Notes in Artificial Intelligence, vol. 827. Heidelberg, Germany: Springer; 1994, p. 480–505.

[5] Noda I, Nakashima H, Handa K. Programming language gaea and its application for multiagent systems. In: Workshop on Multi-Agent System and Logic Programming, December 1999.

[6] Dix J. The logic programming paradigm. AI Communications 1998;11(3):39–43 (Short version in Newsletter of ALP 1998;11(3):10–14).

[7] Poggi A. Daisy: an object-oriented system for distributed artificial intelligence. In: Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures, and Languages, 1994.

[8] Van Roy P, Haridi S. Mozart: a programming system for agent applications. In: International Workshop on Distributed and Internet Programming with Logic and Constraint Languages, November 1999. Part of International Conference on Logic Programming (ICLP 99).

[9] Yamazaki K, Yoshida M, Amagai Y, Takeuchi I. Implementation of logic computation in a multi-paradigm language tao. Information Processing Society of Japan 2001;41(1):142–57.

[10] Lee JHM, Pun PKC. Object logic integration: a multiparadigm design methodology and a programming language. Computer Languages 1997;23(1):25–42.

[11] Ng KW, Huang L, Sun Y. A multiparadigm language for developing agent-oriented applications. In: Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS), Beijing, China, September. New York: IEEE; 1998.
[12] Fayad ME, Schmidt DC. Object-oriented application frameworks (special issue introduction). Communications of the ACM 1997;40(10):39–42.
[13] Fayad ME, Johnson R, editors. Domain-specific application frameworks: frameworks experience by industry. New York: Wiley, 1999.
[14] O'Keefe RA. Towards an algebra for constructing logic programs. In: Proceedings of the International Symposium on Logic Programming. IEEE Computer Society, Technical Committee on Computer Languages. Rockville, MD: The Computer Society Press, July 1985.
[15] Bugliesi M, Lamma E, Mello P. Modularity in logic programming. The Journal of Logic Programming 1994;19& 20:443–502.
[16] Zunino A, Campo M, Mateos C. Simplifying mobile agent development through reactive mobility by failure. In: Bittencourt G, Ramalho G, editors, Advances in artificial intelligence. Lecture Notes in Computer Science, vol. 2507. Berlin: Springer; Novermber, 2002. p. 163–174.
[17] Lange DB, Oshima M. Seven good reasons for mobile agents. Communications of the ACM 1999;42(3):88–9.
[18] Gray RS, Cybenko G, Kotz D, Rus D. Mobile agents: motivations and state of the art. In: Bradshaw J, editor. Handbook of agent technology. Menlo Park, USA/Cambridge, MA: AAAI/MIT Press; 2001.
[19] Bradshaw JM. Software agents. Menlo Park, USA: AAAI Press; 1997.
[20] Garcia A, Chavez C, Silva O, Silva V, Lucena C. Promoting advanced separation of concerns in intra-agent and inter-agent software engineering. In: Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA'2001, October 2001.
[21] Cordero D, Roldán P, Schiaffino S, Amandi A. Intelligent agents generating personal newspapers. In: Proceedings of the International Conference on Enterprise Information Systems, Portugal, March 1999.
[22] Schiaffino S, Amandi A. User profiling with case-based reasoning and bayesian networks, in: Proceedings of the Seventh Iberoamerican Conference on Artificial Intelligence (IBERAMIA 2000) and the 15th Brazilian AI Symposium (SBIA 2000), Atibaia, São Paulo, Brasil, November 2000.
[23] Zunino A, Amandi A. Building multi-agent systems from reusable software components. In: Alvares LO, editor. Proceedings of the Third Workshop in Distributed Artificial Intelligence and Multi-Agent Systems (3WDAIMAS) held in conjunction with the Seventh Iberoamerican Conference on Artificial Intelligence (IBERAMIA 2000) and the 15th Brazilian AI Symposium (SBIA 2000), Atibaia, São Paulo, Brasil, November 2000.
[24] Reticular Systems Inc., AgentBuilder: an integrated toolkit for constructing intelligent software agents, White Paper, February 1999. http://www.agentbuilder.com.
[25] Busetta P, Ramamohanarao K. The BDIM agent toolkit design. Technical Report 97/15, Departament of Computer Science, University of Melbourne, 1997.
[26] Nwana H, Ndumu D, Lee L, Collis J. ZEUS: a tool-kit for building distributed multi-agent systems. Applied Artifical Intelligence Journal 1999;13(1):129–86. http://www.labs.bt.com/projects/agents/zeus/index.htm.
[27] Denti E, Omicini A. 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), New Mexico, USA, December 1999.
[28] Kellett A, Fisher M. Coordinating heterogeneous components using executable temporal logic. In: Meyer J-J, Treur J, editors. Agents, reasoning and dynamics, Series of Handbooks in Defeasible Reasoning and Uncertainty Management Systems, vol. 6. Dordecht: Kluwer Academic Publishers: 2001.
[29] Weerasooriya D, Rao A, Ramamohanarao K. Design of a concurrent agent-oriented language. In: Wooldridge MJ, Jennings NR, editors. Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures and Languages: Intelligent Agents I, Lecture Notes in Artificial Intelligence, vol. 890. Berlin: Springer, August 1995, 386pp.
[30] Levesque HJ, Reiter R, Lespérance I, Lin F, Scherl RB. GOLOG: a logic programming language for dynamic domains. Journal of Logic Programming 1997;31(1–3):59–83.
[31] Tarau P. Jinni: a lightweight java-based logic engine for internet programming. In: Sagonas K, editor. Proceedings of JICSLP'98 Implementation of LP languages Workshop, Manchester, UK, June 1998, invited talk.
[32] Graham J, Decker K. Towards distributed, environment centred agent framework. In: Proceedings of Agent Theory and Languages (ATAL) '99, July 1999.

[33] Kendall EA, Krishna PVM, Pathak CV, Suresh CB. A framework for agent system. In: Fayad ME, Schmidt DC, Johnson RE, editors. Implementing applications frameworks: object oriented frameworks at work. New York: Wiley; 1999.

[34] Lange DB, Oshima M. Programming and deploying mobile agents with java aglets. Reading, MA, USA: Addison-Wesley; September 1998.

[35] Weld DS. An introduction to least commitment planning. AI Magazine 1994;15(4):27–61.

[36] Amandi A, Iturregui R, Zunino A. Object-agent oriented programming, Electronic Journal of Sociedad Argentina de Informática e Investigación Operativa (EJS) also in Second Argentine Symposium on Object Orientation (ASOO'98) 1999; 2(1):5–16. ISSN 1514-6774.

[37] Teyseyre A. A 3d visualization approach to validate requirements. In: Congreso Argentino de Ciencias de la Computación, October 2002.

Analía Amandi received a Ph.D. Degree in Computer Science from the Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil in 1997 and the Computer Science degree from the UNLP University, La Plata, Argentina in 1990. Currently she is a Professor at Computer Science Department and head of the agent group of the ISISTAN Research Institute of the UNICEN University at Tandil, Argentina. She has over 30 papers published in conferences and journals about agents. Her research interests includes interface agents and software architecture.

Marcelo Campo received a Ph.D. Degree in Computer Science from the Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil in 1997 and the Systems Engineer degree from the UNICEN University, Tandil, Argentina in 1988. Currently he is an Associate Professor at Computer Science Department and Head of the ISISTAN Research Institute of the UNICEN University at Tandil, Argentina. He is also a research fellow of the National Council for Scientific and Technical Research of Argentina (CONICET). He has over 50 papers published in main conferences and journals about software engineering topics. His research interests includes intelligent aided software engineering, software architecture and frameworks, agent technology and software visualization.

Alejandro Zunino received a Ph.D. Degree in Computer Science from the Universidad Nacional del Centro (UNICEN), Tandil, Argentina, in 2003, his MSc. in Systems Engineering in 2000 and the Systems Engineer degree in 1998. He is a full time research assistant at the UNICEN. He has published over 15 papers in journals and conferences. His current research interest focus on development tools for intelligent agents, mobile agents and logic programming.