This is a preprint of the article: "A. Zunino, M. Campo and C. Mateos: "Reactive Mobility by Failure: When Fail Means Move". Information Systems Frontiers. Special Issue on Mobile Computing and Communications: Systems, Models and Applications. Springer Science + Business Media B.V. Vol. 7, Number 2, pp. 141-154. ISSN 1387-3326. 2005."

The original publication is available at http://dx.doi.org/10.1007/s10796-005-1475-2

Reactive Mobility by Failure: When Fail Means Move

Alejandro Zunino Marcelo Campo* Cristian Mateos†

ISISTAN Research Institute - UNICEN University
Campus Universitario (B7001BBO), Tandil, Bs. As., Argentina
TEL: +54 (2293) 440363 int. 35
FAX: +54 (2293) 440363 int. 52
azunino@exa.unicen.edu.ar

*Also National Council for Scientific and Technical Research of Argentina (CONICET).
†Also Council for Scientific Research of Buenos Aires, Argentina (CIC).

This is a preprint of the article: "A. Zunino, M. Campo and C. Mateos: "Reactive Mobility by Failure: When Fail Means Move". Information Systems Frontiers. Special Issue on Mobile Computing and Communications: Systems, Models and Applications. Springer Science + Business Media B.V. Vol. 7, Number 2, pp. 141-154. ISSN 1387-3326. 2005."

The original publication is available at http://dx.doi.org/10.1007/s10796-005-1475-2

Abstract

Mobile agent development is mainly supported by Java-based platforms and tools. However, the weak mobility model they use, the lack of adequate support for developing inference and reasoning mechanisms, added to the inherent complexity of building location-aware software, impose strong limitations for developing mobile intelligent agent systems. In this article we present MoviLog, a platform for building Prolog-based mobile agents with a strong mobility model. MoviLog is an extension of JavaLog, an integration of Java and Prolog, that allows us to take advantage of the best features of the programming paradigms they represent. MoviLog agents, called Brainlets, are able to migrate among different Web sites, either proactively or reactively, to use the available knowledge in order to find a solution. The major contribution of MoviLog is its *Reactive Mobility by Failure* (RMF) mechanism. RMF is a mechanism that acts when an agent needs a resource or service that is not available at the current executing site. RMF uses a distributed multi-agent system to transparently transport the executing agent to the site where the resource/service is available, thus reducing the development effort with respect to the traditional mobile agent approach, while maintaining its advantages.

Keywords: mobility, mobile agents, intelligent agents, logic programming

1 Introduction

Many researchers envision the future Web as a global community where people and intelligent agents interact and collaborate (Hendler, 2001), sharing interests as well as resources. In this scenario consisting of sites providing highly dynamic content, mobile users using unreliable connections and small devices such as PDAs and cell phones, mobile agents will have a fundamental role (Gray et al., 2001). A mobile agent is a computer program which represents a user in a computer network and is capable of migrating autonomously between hosts to perform some computation on behalf of the user. Such a capability is particularly interesting when an agent makes sporadic use of a valuable shared resource. Also, efficiency can be improved by moving agents to different hosts to query a large database. This approach may also improve the response time and the availability when compared to performing the interactions over network links that are subject to long delays or interruptions.

Despite the well known advantages of mobile agents (Lange and Oshima, 1999), their usage is still limited to small applications, mainly due to the following factors:

Development effort: mobile agents are inherently more complex than traditional stationary systems. Clearly, mobile agent developers have to provide mechanisms to decide an agent's itinerary. Therefore, though agents' location awareness may be very beneficial, it also adds further complexity to the development of intelligent mobile agents, specially with respect of stationary applications (Picco et al., 1997; Silva et al., 2001).

Lack of standards for accessing resources, including legacy systems: there is a need for developing mechanisms to allow agents to access resources offered by sites. In particular, mobile agent technology should be capable of using existent resources such as Web pages and Web-accessible programs and devices.

Security concerns: this has been deeply studied, and some good results have been achieved (Gray et al., 1998; Tripathi et al., 2002). Though recent platforms have shown that it is perfectly feasible to build secure mobile agent systems, there are still psychological reasons against mobile agents (Wagner and Turban, 2002), thus we will limit the scope of the paper to the first two points.

In this paper we describe MoviLog, a platform for mobile agents that aims at reducing the development effort by automating decisions on *when* and *where* to move an agent, based

on resource needs. MoviLog is an extension of the JavaLog framework (Amandi et al., 1999; Zunino et al., 2001) which implements an extensible integration between Java and Prolog (see Appendix A for an introduction to Prolog).

MoviLog provides mobility enabling mobile logic-based agents, called Brainlets, to migrate between Web sites by using a *strong mobility* mechanism. Strong mobility implies that an agent's execution state is transferred and resumed at the remote site. On the other hand, *weak mobility*, the mechanism implemented by most Java-based mobile agent platforms, is not able to transfer the execution state thus the agent *forgets* the point at where its execution was. Besides extending Prolog with operators to implement proactive strong mobility, the most interesting aspect of MoviLog is the incorporation of the notion of *reactive mobility by failure* (RMF). This mechanism acts when a specially declared Prolog goal fails, by transparently moving a Brainlet to another Web site to satisfy its resource and service needs.

The article is structured as follows. The next section briefly describes the JavaLog framework. Section 3 introduces the MoviLog platform. Section 3.3 presents the integration of MoviLog with Web services. In Section 3.4 the evaluation algorithm is briefly described. Section 4 reports some experimental results. Then, Section 5 discusses the most relevant related works. Finally, in Section 6 concluding remarks and future works are presented.

2 The JavaLog Framework

Intelligent agents are usually developed by using general purpose object oriented languages such as C++ or Java due to the advantages that encapsulation and inheritance offer (Crnogorac et al., 1997). Despite these advantages, object-oriented languages do not provide specific abstractions for building agents such as reasoning mechanisms, inference, learning or knowledge representation. As a consequence, developers are forced to use programming abstractions not well suited for agents or hand code these mechanisms before building agents.

On the other hand, a logic-oriented programming approach is a straightforward consequence of the requirement of managing knowledge and reasoning. For this reason, logic languages such as Prolog (Nilsson and Maluszynaski, 1995), Agent0 (Shoham, 1997), Metatem (Fisher, 1994) and Gaea (Noda et al., 1999) are considered good alternatives for programming intelligent agents (Dix, 1998).

Certainly, multi-paradigm languages integrating both logic and object-oriented paradigms are convenient choices for the definition of agent programming languages because they

offer the best of both worlds. Examples of multi-paradigm languages appropriate for programming agents are discussed in (Van Roy and Haridi, 1999; Yamazaki et al., 2001; Lee and Pun, 1997; Ng et al., 1998; Amandi et al., 1999).

JavaLog is a multi-paradigm language that integrates Java and Prolog (Amandi et al., 1999; Zunino et al., 2001). The JavaLog support is based on an extensible Prolog interpreter designed as a framework (Fayad and Johnson, 1999). This means that the basic Prolog engine can be extended to accommodate different extensions, such as multi-threading, modal logic operators, or mobility, for example.

JavaLog defines the *logic module* (a list of Prolog clauses) as its basic concept of manipulation. In this sense, both objects and methods from the object-oriented paradigm are considered as modules encapsulating data and behavior, respectively. The elements manipulated by the logic paradigm are also mapped to modules.

Each agent possesses an object called *brain*. This object is an instance of an extended Prolog interpreter implemented in Java which enables developers to use objects within logic clauses, as well as to embed logic modules within Java code. In this way, each agent is an instance of a class that can define part of its methods in Java and part in Prolog. The definition of a class can include several logic modules defined within methods as well as referenced by instance variables.

The JavaLog language defines some interaction constraints between object-oriented and logic modules. These interaction constrains are classified by referring, communication and composition constraints. Referring constraints specify the composition limits of different modules. Communication constraints specify the role of objects in logic modules and the role of logic variables in methods. Composition constraints specify how logic modules can be combined.

3 The MoviLog Platform

MoviLog is an extension of the JavaLog framework to support mobile agents on the Web. MoviLog implements strong mobility for a special type of agents called *Brainlets*. The MoviLog inference engine is able to process several concurrent threads and to restart the execution of an incoming Brainlet at the point where it migrated, either pro-actively or reactively, in the original host. Some early ideas about MoviLog are described in (Zunino et al., 2002). This paper is mainly focused on the novel mobility mechanism supported by MoviLog and the conceptual model behind it.

In order to enable mobility across sites, each Web server belonging to a MoviLog network have to be extended with a MARlet (Mobile Agent Resource). A MARlet is a

Java servlet encapsulating the MoviLog inference engine and providing services to access it through the Web. 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 modules, and answering logic queries. In this sense, a MARlet can also be used to provide inferential services to legacy Web applications or agents.

From the mobility point of view, MoviLog provides support to implement Brainlets with typical pro-active capabilities, but more interesting yet, it implements a mechanism for transparent reactive mobility by failure (RMF). This support is based on a number of stationary agents distributed across the network. These agents provide intelligent mechanisms to automatically migrate Brainlets based on their resource requirements. Further details on this will be explained in Section 3.2.

3.1 Proactive Strong Mobility

MoviLog strong mobility mechanism allows a Brainlet to proactively migrate its execution. Migration is achieved by invoking a MoviLog predicate moveTo(S), where S is the destination site. The migration mechanism implemented by moveTo works as follows. Before transport, MoviLog serializes the Brainlet and its execution state - i.e. its knowledge base and code, current goal to satisfy, instantiated variables, choice points, etc. Then, it sends the serialized form to its counterpart on the destination host. In the remote host, MoviLog reconstructs the Brainlet and its execution state, and then its execution is resumed. Eventually, after performing some computation, the Brainlet can return to the originating host by calling the return predicate.

The following example presents a simple Brainlet for e-commerce whose goal is to of find and buy an article in the network according to a number of preferences provided by a user. The buy clause looks for offers available in different sites of the network, selects 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 through a number of 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:

```
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]):-
```

```
    moveTo(S), article( A, Offer, URL), O= (S,Offer,URL),
   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)),
    moveTo(S), buy_article(O,E),  return.
?- buy(#Art).
```

Although proactive mobility provides a powerful tool to take advantage of network resources, in the case of Prolog, it also adds extra complexity due to its procedural nature. 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*.

3.2 Reactive Mobility by Failure

Intelligent agents have been traditionally considered as systems possessing several dimensions of attributes (Nwana, 1996; Shoham, 1997; Genesereth and Ketchpel, 1994). For example, (Bradshaw, 1997) described mobile intelligent agents in terms of a three dimensional space defined by *agency*, *intelligence* and *mobility*: *agency* is the degree of autonomy and authority vested in the agent; *intelligence* is the degree of reasoning and learning behavior; *mobility* is the degree to which agents themselves travel through the network.

Based on these views we consider a mobile agent as composed of two separated and orthogonal behaviors: stationary behavior and mobile behavior; the first one is concerned with the tasks performed by an agent on specific places of the network, and the second one is in charge of making decisions about mobility. The MoviLog platform provides a new form of mobility called Reactive Mobility by Failure (RMF) that is based on the idea that those two functionalities or concerns can be treated independently at the implementation level (Garcia et al., 2001).

RMF aims at reducing the effort of developing mobile agents by automating some decisions about mobility. In this way, programmers focus their efforts on the stationary functionality of mobile agents, and delegate mobility on RMF.

RMF is a reactive migration mechanism able to automate decisions on when and where to migrate a Brainlet based on its resource needs. Reactive migration mechanism are based on the idea that an entity external to the Brainlet triggers mobility. In RMF those *external entities* are stationary agents (not able to move between sites) that are part of the

MoviLog platform thus provide runtime support for RMF. Stationary agents interfere with the normal execution of a Brainlet when a *failure* occurs.

Conceptually, a failure occurs when a Brainlet tries to access a resource that is located at a remote site. In terms of Prolog, a failure occurs when a Brainlet evaluates a specially declared goal that cannot be deduced from the clauses provided by the local site, as depicted in the step i of Fig. 1. At this point, the stationary agents obtain a list of sites to try to probe the goal and migrate the Brainlet to one of these sites (ii of Fig. 1). In addition, stationary agents may build an itinerary for the Brainlet in order to visit the sites according to some policy.

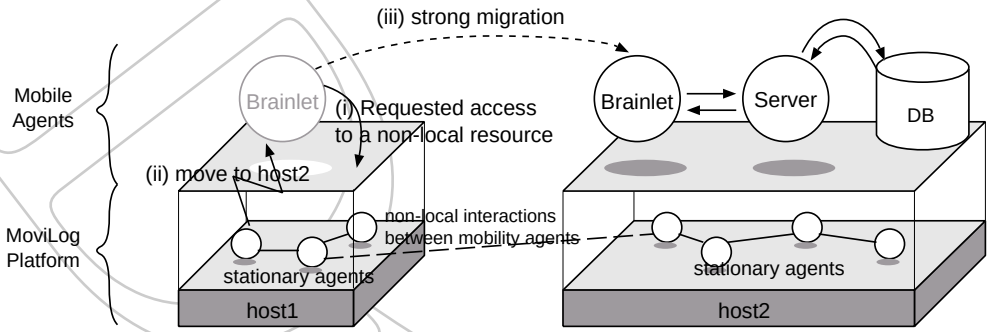

Figure 1: Reactive Mobility by Failure

To sup up, RMF is able to automate two types of decisions about mobility:

- *where to migrate*: RMF selects the next destination of the mobile agent. This decision can take into account factors such as network traffic, CPU load, etc. Moreover, it may be necessary to visit several sites in a certain sequential order. For example, *host1*, *host2* and *host3*, or *host2*, *host1* and *host3*. A problem here is that Brainlets might not know the location of resources. As a consequence RMF has to manage information about resources provided by sites capable of executing Brainlets.

- *when to migrate*: in the example, when a failure occurs, RMF migrate the mobile agent to *host2*. However, it could be convenient to copy the required clauses from *host2* to *host1*. Those decisions can be taken considering network traffic and time.

As shown in Fig. 1, RMF consists of:

- executing units or mobile agents called *Brainlets*.

- mechanisms enabling Brainlets to specify on which resources RMF is allowed to act.
- a platform supporting the execution of Brainlets and responsible for taking decisions about mobility.

Each of these elements is described in detail in the following sections.

3.2.1 Brainlets

A Brainlet is composed of the following parts:

- *code and data*: consist of Prolog clauses and Java objects implementing a Brainlet's behavior and knowledge.
- *execution state*: one ore more threads encompassing a program counter, stack, variable bindings and choice points. It is worth noting that these threads persist after migrating an agent to a remote host.
- *protocol clauses*: are used to delegate mobility decisions on RMF.

The code of a Brainlet consists of two sections: Protocols and Clauses. This first section contains protocol declarations. The second section contains clauses expressed in JavaLog. Syntactically, the code of a Brainlet has the following form:

PROTOCOLS

...

CLAUSES

...

The next section is concerned with protocols and its usage.

3.2.2 Protocols

A protocol describes an interface used to access a resource available at any site of the network. When a Brainlet accesses one of these resources, and the resource is not available at the local site, the stationary agents are activated.

In terms of Prolog, protocol declarations have the syntax protocol(functor, arity). Such a declaration enables RMF to act on failures of goals with the form:

functor(<argument1>, <argument2>, ..., <argumentArity>).

This is a preprint of the article: "A. Zunino, M. Campo and C. Mateos: "Reactive Mobility by Failure: When Fail Means Move". Information Systems Frontiers. Special Issue on Mobile Computing and Communications: Systems, Models and Applications. Springer Science + Business Media B.V. Vol. 7, Number 2, pp. 141-154. ISSN 1387-3326. 2005."

The original publication is available at http://dx.doi.org/10.1007/s10796-005-1475-2

In this way, when a goal declared as a protocol by a Brainlet fails, stationary agents transparently move the Brainlet to another site having definitions for such a protocol. Thereafter it continues the normal execution to try to solve the goal.

The following code shows the implementation of the customer agent combining both mobility mechanisms. This solution collects through backtracking the matching articles from the database until no more articles are left. The *article* protocol makes the Brainlet to try all the sites offering the same protocol before returning to the origin site to collect (by using 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. As can be noted, the solution using RMF looks much like a common Prolog program. Certainly, this solution is simpler than the one using just proactive mobility.

```
PROTOCOLS
  protocol(article, 3).
CLAUSES
  preference(car, [ford, Model, Price]) :- Model > 1998, Price < 20000.
  preference(tv,[sony, Model, Price]) :- Model = 21in, Price < 1500.
  lookForOffers(A, [O|RO], [O|Roff]) :- article( A, Offer, URL),
    thisSite(ThisSite), assert( offer(ThisSite, Offer, URL)), fail.
  lookForOffers(A, _, Offers) :- !, findAll(offer(S,O,E)), Offers).
  buy(Art) :- lookForOffers(Art,R,Offers), selectBest(Offers,(S,O,E)),
    moveTo(S), buy_article(O, E),  return.
...
?- buy(Art).
```

It is worth noting that the protocol article is only a description of a resource. The concrete realization of the resource may be a set of Prolog clauses, a Java method, a relational database or a Web service. MoviLog abstracts the complexity of accessing these resources by providing a simple access method.

RMF allows a programmer to adapt and extend the different decision mechanisms that act when a failure is detected. For example, it is possible to use several pre-defined or user-defined algorithms and metrics for building and updating an agent's itinerary.

3.2.3 RMF Execution Support

RMF is implemented through a Multi-Agent System (MAS) composed of stationary agents in charge of mobility. It is worth noting that these agents act only when a failure occurs on a goal declared as a protocol. The MAS consists of two types of stationary agents:

- *Protocol Name Servers (PNS)*: Each host capable of executing Brainlets has one PNS. PNS are responsible for managing information on protocols offered at each site. When a Brainlet needs a resource not hosted at the local site, RMF queries the PNS of the site to obtain a list of sites offering that resource. A site offering resources registers with its PNS the protocols of the resources. Then, the PNS announces the new protocols to other sites by using a multicast-based communication mechanism.

- *Mobility agents*: Mobility agents collaborate with PNSs in order to select the next destination of a Brainlet that failed, building an itinerary when a resource is hosted in more that one site. All these decisions take into account policies based on network traffic, link speed or user-defined metrics. In addition, mobility agents are able to decide whether to migrate an agent or to transfer a resources from other sites to the site where the Brainlet is located.

3.3 RMF and Web Services

Let us suppose a scenario consisting of a Notebook with a low bandwidth connection to a network. Servers S_1,S_2,..., S_n residing in this network have good connectivity with the Internet (Fig. 2). In addition, these servers are able to execute Brainlets. A Brainlet running at the Notebook requires accessing to a service offered by a server P located at the Internet. Due to connectivity constraints, it might be advantageous to migrate the Brainlet to P. However, P does not support the execution of Brainlets, so this approach is unfeasible. An alternative is to migrate the Brainlet to one of the servers S_i with good connectivity. From there, the Brainlet will interact with P by using a fast network link, returning to the Notebook afterwards.

MoviLog supports interaction with Web Services to enable the usage of RMF in cases where the site hosting a required service cannot execute Brainlets. *Web Services* (Vaughan-Nichols, 2002) – Web-accessible programs and devices – can be viewed as a set of programs interacting cross a network with no explicit human interaction during the transaction. In order for programs to exchange data, it is necessary to define the communications protocol, the data transfer syntax, and the location of the endpoint. For building large, complex systems, such service definitions must be done in a rigorous manner: ideally, a machine-readable language with well-defined semantics, as opposed to parochial and imprecise natural languages.

The Web Services Description Language (WSDL) is an XML-based language for describing Web services as a set of network endpoints that operate on messages. A WSDL

This is a preprint of the article: "A. Zunino, M. Campo and C. Mateos: "Reactive Mobility by Failure: When Fail Means Move". Information Systems Frontiers. Special Issue on Mobile Computing and Communications: Systems, Models and Applications. Springer Science + Business Media B.V. Vol. 7, Number 2, pp. 141-154. ISSN 1387-3326. 2005."

The original publication is available at http://dx.doi.org/10.1007/s10796-005-1475-2

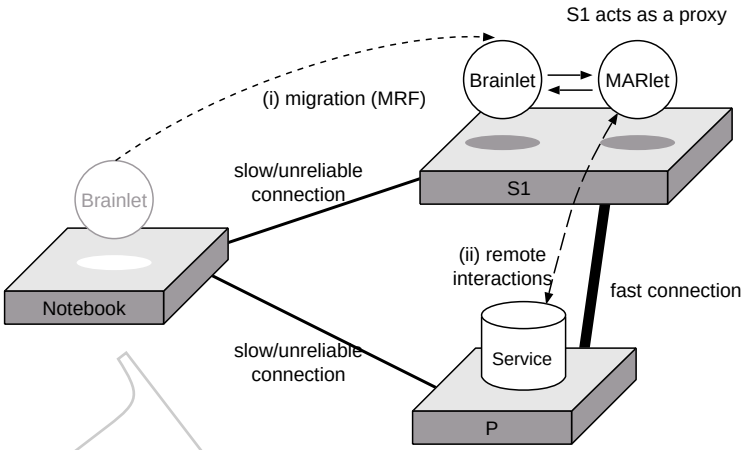

Figure 2: MARlet acting as a proxy of a Web accessible service

service description contains an abstract definition for a set of operations and messages, a concrete protocol binding for these operations and messages, and a network endpoint specification for the binding.

From a WSDL description a program can determine the services provided by a server and how to invoke and use these services, independently of the network protocol or programming language. As a complement to WSDL, the WWW Consortium developed the Universal Description, Discovery and Integration (UDDI) specification. UDDI provides a method for publishing and finding service descriptions written in WSDL or any other service description mechanism.

In order to integrate MoviLog with Web services, we extended the PNS agents with capabilities of querying UDDI registries, parsing WSDL documents and mapping Prolog clauses to and from Web services. When a PNS detects a failure, it first obtains the resources directly available as Prolog predicates, then it queries a UDDI registry and obtains a list of Web-accessible services that match the protocol of the predicate causing a failure. When the mobile agent effectively tries to access to a Web-accessible resource, a PNS agent determines whether to travel to the remote site or not, depending on its support for mobile agents, network load, size of the mobile agent, etc.

Let us consider the usage of Web services not registered in UDDI. A Brainlet can invoke Web services by using a special code section named *services* which contains, among other things:

- pointers to WSDL descriptions of the services, including their names, parameters

and access points. For example, *registerServices*('`http://soap.amazon.com/schemas/AmazonWebServices.wsdl`') registers the services provided by Amazon (*amazonComKeywordSearchRequest*, *amazoncomAuthorSearchRequest*, etc.) and associates these services to Prolog clauses.

- a number of user-defined clauses to adapt a protocol clause to a Web service. This involves at least the following steps:
  - given a protocol clause and its parameters, the programmer has to map these parameters to the input parameters of a given Web service.
  - the output of the Web service has to be converted to Prolog structures. MoviLog assumes that Web services output XML. MoviLog provides two alternatives: the usage of XPath[1] to query the XML output, or the usage of XSL Transformations to transform the XML output to Prolog terms.

Let us consider, for example, the keyword search service provided by Amazon. The WSDL[2] description of the service *KeyWordSearchRequest* describes its parameters and types. The most important parameters are *keywords* and *type* (brief or detailed response). In addition a WSDL document includes a description of the invocation method of the service. In this case, the service has to be invoked through a HTTP SOAP-encoded request to the URL `http://soap.amazon.com/onca/soap`. By knowing this details it is easy for MoviLog to invoke *KeyWordSearchRequest* or any other service. Now, the previous example of RMF can be extended to search and buy books in both Amazon and the MoviLog network:

```
SERVICES
  registerServices('http://soap.amazon.com/-schemas/-Amazon-Web-
Services.wsdl').
  articleAdapter(book(FeatureList), Offers, URLs) :-
    % invokes the Web service
    amazoncomKeywordSearchRequest( [keyword:FeatureList, mode:books,
      type:lite, format:xml, devtag: 'D26UGIDJJ9HCRX'], XMLresponse ),
    % transforms the XML response by using a XSL transformation
```

[1] XPath (Gottlob et al., 2003) is a language for addressing parts of an XML document.

[2] The Amazon.com Web Services software development kit is available at `http://associates.amazon.com/exec/panama/associates/join/developer/kit.html`.

```
    transformWith( XMLresponse, 'lite-data-to-prolog.xsl', Offers ),
    % uses XPath to obtain the URLs of the books
    xpath(XMLresponse, '//ProductInfo/Details@url'), URLs).
PROTOCOLS
  protocol(article, 3).
CLAUSES
  preference(book, [author:A, keywords:[mystery, terror], price:P]) :-
    (A == 'Stephen King' ; A == 'Agatha Christie'), P < 20.
  lookForOffers(A, [O|RO], [O|Roff]) :- article( A, Offer, URL),
    thisSite(ThisSite), assert( offer(ThisSite, Offer, URL)), fail.
  lookForOffers(A, _, Offers) :- !, findAll(offer(S,O,E)), Offers).
 buy(Art) :- lookForOffers(Art,R,Offers), selectBest(Offers,(S,O,E)),
    moveTo(S), buy_article(O, E),  return.
...
?- buy(Art).
```

To sum up, in order to allow Brainlets to interact with legacy systems not able to run mobile agents, MoviLog is able to invoke Web services. To do so, developers have to provide a set of Prolog clauses that map a protocol description to a Web service and a method to extract the information from the response.

3.4 Evaluation Algorithm

In this section we briefly describe the evaluation algorithm used by MoviLog. RMF can be understood by considering a classical Prolog interpreter with a stack S, a database D, and a goal g. Each entry of S contains a reference to the clause c being evaluated, a reference to the term of c that is being proved, a reference to the preceding clause and a list of variables and their values in the preceding clause to be able to backtrack. MoviLog extends this structure by adding information about the distributed evaluation mechanism. The idea is to keep a history of visited MARlets and possibilities for satisfying a given goal within a MARlet.

Protocol definitions create the notion of a *virtual database* distributed among several Web sites. When a Brainlet defines a given protocol predicate in a MARlet h_n, MoviLog informs the PNS agents, which in turn inform the rest of registered MARlets that the new protocol is available in h_n. 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 clauses stored in a remote MARlet with the same protocol clause as the current goal g. Now, in order to probe g the

interpreter has to try with 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 transfers 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 trying to probe the goal. 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 left, the Brainlet is moved to the site from which the Brainlet originated.

To better understand these ideas, let us give a more precise description of the evaluation mechanism. Let $s = \langle c, t_i, V, H, L \rangle$ be an element of the stack, where $c = h : -t_1, t_2, \ldots, t_n$ is the clause being evaluated, t_i is the term of c being evaluated, V is a set of variable substitutions (ex. $X = 1, X = Z$) and $H = \langle H_t, H_v, P \rangle$, where H_t is a list of MARlets not visited, H_v is a list of MARlets visited and P is a list of candidate clauses at a given MARlet that match the protocol clause of c; and L is a list of clauses with the same name and arity as t_i (candidate clauses at the local database).

The interpreter has two states: *call* and *redo*. When the interpreter is in state *call*, it tries to probe a goal. On the other hand, in state *redo* it tries to search for alternative ways of evaluating a goal after the failure of a previous attempt. Given a goal $? - t_1, t_2, \ldots, t_n$, $S = \{\}$ and $state = call$,

1: **if** $state == call$ **then**
2: the interpreter pushes into the stack:

$$\langle t_1, t_2, \ldots, t_n, t_i, V = \{\}, \langle H_t = \langle\rangle, H_v = \langle\rangle, PH_t = \langle\rangle \rangle$$

3: **for all** i such that $1 \leq i \leq n$ **do**
4: **if** The MARlet is visited for the first time **then**
5: the interpreter searches into the local database for clauses with the same name and arity as t_i. This result is stored into P (a list of clauses c_j at the current MARlet).
6: **else**
7: P is updated with the clauses available at the current MARlet.
8: **end if**
9: Then, the more general unifier (MGU) for t_i and the head of c_j is calculated. If there is no such unifier for a given c_j, then c_j is removed from P. Otherwise, the substitutions for t_i and the head of c_j are stored into V. At this point, the

algorithm tries to probe c_j by jumping to line 1. If every t_i is successfully proved, then the algorithm returns *true*.

10: If there is not a clause c_j such as there is a more general unifier for t_i and the head of c_j, the interpreter queries a PNS for a list of MARlets offering the same protocol clause as t_i. This is stored into H_t. Then, the Brainlet is moved to the first MARlet h_d in H_t. The current MARlet is removed from H_v to avoid visit it again.

11: If H_v is empty then $state = redo$

12: **end for**

13: **else**

14: This point of the execution is reached when the evaluation of a goal fails at the current MARlet. The step 9 of the algorithm selected a c_j from the local database for proving t_i. This selection was the source of the failure. Therefore, MoviLog simply restores the clause by reversing the effects of applying the substitutions in V, selects another clause c_j, sets $state = call$ and jumps to line 4.

15: If there are no more choices left in P, this implies that it is not possible to prove t_i from the local database. Therefore the top of the stack is popped and the algorithm returns false. This may require migrating the Brainlet to the site where the goal failed for the first time.

16: **end if**

3.4.1 Distributed Backtracking and Consistency Issues

The RMF model generates several tradeoffs related to the standard Prolog execution semantics. Backtracking is one of them. When a Brainlet moves around several places, many backtracking points can be left untried, and the question is how the backtracking mechanism should proceed. The solution adopted by MoviLog at the current version resides in the stationary agents. These agents provide a sequential view of the multiple choice points that 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 defines how the local view of a Brainlet is updated when it arrives to a host. This involves automatically querying the PNS agents to obtain a list of MARlets implementing a given protocol clause and querying the current MARlet in order to obtain a list of clauses matching the protocol

clause being evaluated.

4 Experimental Results

In this section we report the results obtained with an application implemented by using MoviLog, μCode (Picco, 1998) (a Java-based framework for mobile agents) and Jinni (Tarau, 1998) (a Prolog-based language with support for strong mobility).

The application consists of a number of customer agents that are able to select and buy articles offered by sellers based on users' preferences. Both, customers and sellers reside in different hosts of a network. In this example, customers are ordered to buy books that has to satisfy a number of preferences such as price, author, subject, etc.

The implementation of the application with MoviLog using RMF was easy (39 lines of code). On the other hand, to develop the application by using μCode we had to provide support for representing and managing users' preferences. The size of the application was 741 lines of code[3]. Finally, the Jinni implementation was easier, although not as easy as with MoviLog, due to the necessity of managing agents' code and data closure by hand. The size of the source code in this case was 353 lines. This shows that MoviLog provides more powerful abstractions for developing intelligent mobile agents. From a design point of view, the other platforms intend to be more general, as a consequence their usage for building intelligent agents require more effort.

We tested the implementations on three Pentium III 850 Mhz with 128 MB RAM, running Linux and Sun JDK 1.3.1. To compare the performance of the implementations we distributed a database containing books in the three computers. We ran the agents with a database of 1 KB, 600 KB and 1.6 MB. For each database we ran two test cases varying the user's preferences in order the verify the influence of the number of matched books (state that an agent has to move) on the total running time. On each respective test case the user's preferences matched 0 and 5 books (1 KB database), 3 and 1024 books (600 KB database, 4004 books), and 2 and 1263 (1.6 MB, 11135 books approx.). We ran each test case 20 times and measured the running time. Fig. 3 (right) shows the average running time as a function of the size of the database and the number of products found.

On a second battery of tests we measured the network traffic generated by the agents using the complete database (1.6 MB, 11135 books approx.) distributed across three hosts. Fig. 3 (left) shows the network traffic measured in packets versus the number of books that matched the user's preferences. From the figure we can conclude that MoviLog and its RMF do not affect negatively neither the performance nor the network traffic, while

[3]Not counting the size of a library for handling symbolic user's preferences.

This is a preprint of the article: "A. Zunino, M. Campo and C. Mateos: "Reactive Mobility by Failure: When Fail Means Move". Information Systems Frontiers. Special Issue on Mobile Computing and Communications: Systems, Models and Applications. Springer Science + Business Media B.V. Vol. 7, Number 2, pp. 141-154. ISSN 1387-3326. 2005."

The original publication is available at http://dx.doi.org/10.1007/s10796-005-1475-2

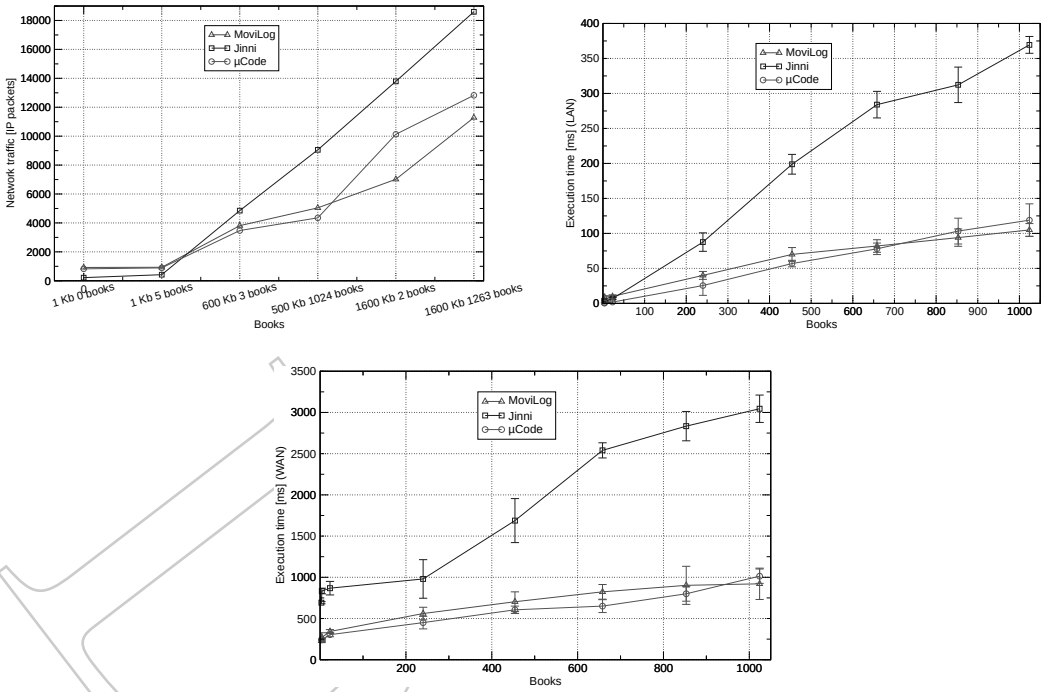

Figure 3: Performance Comparisons

considerably reducing the development effort.

We also ran the application on the Internet by using a configuration of two hosts connected to the same LAN and a third host connected to another LAN. Both LANs were connected through the Internet by using a 256Kbps DSL link. The idea of this test was to evaluate RMF in a real scenario, taking into account the overhead introduced by RMF and its infrastructure. Results on this test are shown in Fig. 3 (bottom). As expected, RMF introduces some overhead due to its PNSs. Despite this overhead, MoviLog is still competitive with the others platforms.

The next section discusses previous work related to MoviLog.

5 Related Work

At present, the only agent programming language that supports interaction with Web services is ConGolog (McIlraith et al., 2001). ConGolog is a model-based programming language that has been used as a testbed for DAML-S (DARPA Agent Markup Language for Web Services) in small applications. The main advantage of MoviLog is its support for reactive mobility by failure that enables us to easily build mobile agents that use Web

services.

With respect to the plethora of work on mobile agents, the most important are the Java-based platforms such as Aglets (Lange and Oshima, 1998), Ajanta (Tripathi et al., 2002) and μCode (Picco, 1998). These systems provide a weak mobility model, forcing a less elegant and more difficult to maintain programming style (Silva et al., 2001). Recent works such as NOMADS (Suri et al., 2000) and WASP (Fünfrocken and Mattern, 1999) extended the Java Virtual Machine (JVM) to support strong mobility. Despite the advantages of strong mobility, these extended JVM do not share some well known features of the standard JVM, such as its ubiquity, portability and compatibility across different platforms.

The logic programming paradigm represents an appropriate alternative to manage agents' mental attitudes. Examples of languages based on it are Jinni (Tarau, 1998) and Mozart/Oz (Haridi et al., 1997). Jinni (Tarau, 1998) is based on a limited subset of Prolog. Jinni lacks adequate support for mobile agents since its notion of code and data closure is limited to the currently executing goal. As a consequence developers have to program mechanisms for saving and restoring an agent's code and data. Mozart (Haridi et al., 1997) is a multi-paradigm language combining objects, functions and constraint logic programming based on a subset of Prolog. Though the language provides some facilities such as distributed scope and communication channels that are useful for developing distributed applications, it only provides rudimentary support for mobile agents.

The main differences between MoviLog and other platforms are its support for RMF, which reduces development effort by automating some decisions about mobility, and its multi-paradigm syntax, which provides mechanisms for developing intelligent agents with knowledge representation and reasoning capabilities integrated with Web services.

6 Conclusion and Future Work

Intelligent mobile agents represent one of the most challenging research areas due to the different factors and technologies involved in their development. Strong mobility and inference mechanisms are, undoubtedly, two important features that an effective platform should provide. MoviLog represents a step forward in that direction. The main contribution of our work is the *reactive mobility by failure* concept. It enables the development of agents using common Prolog programming style, making in it easier thus for Prolog programmers. This concept, combined with proactive mobility mechanisms, also provides a powerful tool for exploiting Web services and mobility.

MoviLog has been used in several research projects. For example, we have devel-

oped a distributed meeting scheduler that uses Brainlets to assist users on managing their calendars. Those Brainlets migrate between sites in order to negotiate meetings with the other users' Brainlets. Each Brainlet manages user preferences including preferred places, contacts, days, etc. These preferences are represented by a Bayesian Network. Another application related to RMF is MOVED, a debugger for mobile agents that supports the concept of RMF. MOVED supports features found in most traditional debugger, but with one very important difference: all these features operate in a distributed and mobile way. This is, breakpoints and watchpoints are set on mobile code, thus MOVED has to take into account mobility issues.

The weakest point of the approach for integrating MoviLog with Web services is that it does not take into account the semantics of Web services. As a consequence, the programmer has to ensure, for example, that a protocol such as *article*, is mapped to Amazon's *KeyWordSearchRequest* or Google's *doGoogleSearch* in the category Shop. The same applies for the arguments and responses. One approach to solve these limitations is the usage of machine understable descriptions of the concepts involved in Web services. We are enriching MoviLog's protocols and services sections with ontologies based on the technologies of the Semantic Web (Berners-Lee et al., 2001).

References

Amandi, A., Zunino, A., and Iturregui, R. (1999). Multi-paradigm languages supporting multi-agent development. In Garijo, F. J. and Boman, M., editors, *Multi-Agent System Engineering*, volume 1647 of *Lecture Notes in Artificial Intelligence*, pages 128–139, Valencia, Spain. Springer-Verlag.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic Web. *Scientific American*, 284(5):34–43.

Bradshaw, J. M. (1997). *Software Agents*. AAAI Press, Menlo Park, USA.

Crnogorac, L., Rao, A. S., and Kotagiri Ramamohanarao (1997). Analysis of inheritance mechanisms in agent-oriented programming. In *Proc. of the* 15^{th} *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 647–654. Morgan Kaufmann Publishers.

Dix, J. (1998). The Logic Programming Paradigm. *AI Communications*, 11(3):39–43. Short version in Newsletter of ALP, Vol. 11(3), 1998, pages 10–14.

Fayad, M. E. and Johnson, R., editors (1999). *Domain-Specific Application Frameworks : Frameworks Experience by Industry*. Wiley & Sons.

Fisher, M. (1994). A survey of concurrent METATEM – the language and its applications. In Gabbay, D. M. and Ohlbach, H. J., editors, *Temporal Logic - Proceedings of the First International Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Heidelberg, Germany.

Fünfrocken, S. and Mattern, F. (1999). Mobile Agents as an Architectural Concept for Internet-based Distributed Applications - The WASP Project Approach. In *Proceedings of KiVS'99 (Kommunikation in Verteilten Systemen)*, pages 32–43. Springer-Verlag.

Garcia, A., Chavez, C., Silva, O., Silva, V., and Lucena, C. (2001). 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*.

Genesereth, M. R. and Ketchpel, S. P. (1994). Software agents. *Communications of the ACM*, 37(7):48–53.

Gottlob, G., Koch, C., and Pichler, R. (2003). XPath processing in a nutshell. *SIGMOD*, 32(2):21–27.

Gray, R. S., Cybenko, G., Kotz, D., and Rus, D. (2001). Mobile agents: Motivations and state of the art. In Bradshaw, J., editor, *Handbook of Agent Technology*. AAAI/MIT Press.

Gray, R. S., Kotz, D., Cybenko, G., and Rus, D. (1998). D'Agents: Security in a multiple-language, mobile-agent system. In Vigna, G., editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag.

Haridi, S., Van Roy, P., and Smolka, G. (1997). An overview of the design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, pages 176–187, Maui, Hawaii, USA. ACM Press.

Hendler, J. (2001). Agents and the semantic web. *IEEE Intelligent Systems*, 16(2):30–36.

Lange, D. B. and Oshima, M. (1998). *Programming and Deploying Mobile Agents with Java Aglets*. Addison-Wesley, Reading, MA, USA.

Lange, D. B. and Oshima, M. (1999). Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89.

Lee, J. H. M. and Pun, P. K. C. (1997). Object logic integration: A multiparadigm design methodology and a programming language. *Computer Languages*, 23(1):25–42.

McIlraith, S., Son, T. C., and Zeng, H. (2001). Semantic web services. *IEEE Intelligent Systems, Special Issue on the Semantic Web*, 16(2):46–53.

Ng, K. W., Huang, L., and Sun, Y. (1998). A multiparadigm language for developing agent-oriented applications. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS)*, Beijing, China. IEEE.

Nilsson, U. and Maluszynaski, J. (1995). *Logic, Programming and Prolog*. John Wiley & Sons, New York, NY.

Noda, I., Nakashima, H., and Handa, K. (1999). Programming language Gaea and its application for multiagent systems. In *Workshop on Multi-Agent System and Logic Programming*.

Nwana, H. (1996). Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244.

Picco, G. (1998). μCode: A Lightweight and Flexible Mobile Code Toolkit. In Rothermel, K. and Hohl, F., editors, *Proceedings of the 2nd International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 160–171. Springer-Verlag: Heidelberg, Germany.

Picco, G. P., Carzaniga, A., and Vigna, G. (1997). Designing distributed applications with mobile code paradigms. In Taylor, R., editor, *Proceedings of the 19th International Conference on Software Engineering*, pages 22–32. ACM Press.

Shoham, Y. (1997). An overview of agent-oriented programming. In Bradshaw, J. M., editor, *Software Agents*, chapter 13, pages 271–290. AAAI Press / The MIT Press.

Silva, A., Romao, A., Deugo, D., and Mira da Silva, M. (2001). Towards a Reference Model for Surveying Mobile Agent Systems. *Autonomous Agents and Multi-Agent Systems*, 4(3):187–231.

Suri, N., Bradshaw, J. M., Breedy, M. R., Groth, P. T., Hill, G. A., Jeffers, R., and Mitrovich, T. S. (2000). An Overview of the NOMADS Mobile Agent System. In 6th *ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages*.

Tarau, P. (1998). Jinni: a lightweight java-based logic engine for internet programming. In Sagonas, K., editor, *Proceedings of JICSLP'98 Implementation of LP languages Workshop*, Manchester, U.K. invited talk.

Tripathi, A. R., Karnik, N. M., Ahmed, T., Singh, R. D., Prakash, A., Kakani, V., Vora, M. K., and Pathak, M. (2002). Design of the Ajanta System for Mobile Agent Programming. *Journal of Systems and Software*. to appear.

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

Vaughan-Nichols, S. J. (2002). Web services: Beyond the hype. *Computer*, 35(2):18–21.

Wagner, C. and Turban, E. (2002). Are intelligent e-commerce agents partners or predators? *Communications of the ACM*, 45(5):84–90.

Yamazaki, K., Yoshida, M., Amagai, Y., and Takeuchi, I. (2001). Implementation of logic computation in a multi-paradigm language TAO. *Information Processing Society of Japan*, 41(1).

Zunino, A., Berdún, L., and Amandi, A. (2001). Javalog: un lenguaje para la programación de agentes. *Inteligencia Artificial, Revista Iberoamericana de I.A.*, 3(13):94–99. ISSN 1337-3601.

Zunino, A., Campo, M., and Mateos, C. (2002). Simplifying mobile agent development through reactive mobility by failure. In Bittencourt, G. and Ramalho, G., editors, *Proceedings of the* 16th *Brazilian Symposium on Artificial Intelligence (SBIA'02)*, volume 2507 of *Lecture Notes in Computer Science*, pages 163–174. Springer-Verlag.

This is a preprint of the article: "A. Zunino, M. Campo and C. Mateos: "Reactive Mobility by Failure: When Fail Means Move". Information Systems Frontiers. Special Issue on Mobile Computing and Communications: Systems, Models and Applications. Springer Science + Business Media B.V. Vol. 7, Number 2, pp. 141-154. ISSN 1387-3326. 2005."

The original publication is available at http://dx.doi.org/10.1007/s10796-005-1475-2

Appendix A: Prolog

Prolog is a logic language that is particularly suited to programs that involve symbolic or non-numeric computation. For this reason it is a frequently used language in Artificial Intelligence where manipulation of symbols and inference about them is a common task.

Prolog, which stands for PROgramming in LOGic, is the most widely available language in the logic programming paradigm. Logic and therefore Prolog is based on the mathematical notions of relations and logical inference. Prolog is a declarative language meaning that rather than describing how to compute a solution, a program consists of a database of facts and logical relationships (rules) which describe the relationships which hold for the given application. Rather than running a program to obtain a solution, the user asks a question. When asked a question, the run time system searches through the database of facts and rules to determine, by logical deduction, the answer.

Among the features of Prolog are *logical variables* meaning that they behave like mathematical variables, a powerful pattern-matching facility called *unification*, a backtracking strategy to search for proofs, uniform data structures, and interchangeable input/output.

Facts

In Prolog we can make some statements by using facts. Facts either consist of a particular item or a relation between items. For example we can represent the fact that it is sunny by writing the program:

```
sunny.
```

Facts can have arbitrary number of arguments from zero upwards. A general model is shown below:

```
relation( <argument1>, <argument2>, ...., <argumentN> ).
```

Relation names must begin with a lowercase letter. For example, the following fact says that a relationship likes links john and mary:

```
likes(john,mary).
```

It is worth noting that names of relations are defined by the programmer. With the exception of a few relations that are built-in, the system only knows about relations that programmers define.

We can now ask a query by asking, for example, *does john like mary?*:

```
?- likes(john,mary)
```

To this query Prolog will answer “yes” because Prolog matches likes(john,mary) in its database.

Variables

How do we say something like *What does Fred eat?* Suppose we had the following fact in our database:

```
eats(fred,apples).
```

To ask what Fred eats, we could type in something like:

```
?- eats(fred,what).
```

However Prolog will say “no”. The reason for this is that *what* does not match with apples. In order to match arguments in this way we must use a *Variable*. The process of matching items with variables is known as *unification*. Variables are distinguished by starting with a capital letter. Thus we can find out what fred eats by typing

```
?- eats(fred,What).
```

Prolog will answer “yes, What=apples”.

Rules

Rules allow us to make conditional statements about our world. Each rule can have several variations, called clauses. These clauses give us different choices about how to perform inference about our world. Let us show an example to make things clearer. Consider the statement *All humans are mortal*. We can express this as the following Prolog rule:

```
mortal(X) :- human(X).
```

The clause can be read as *For a given X, X is mortal if X is human*. To continue let us define a fact *fred is human*:

```
human(fred).
```

If we now pose the question to Prolog ?- mortal(fred). The Prolog interpreter would respond “yes”.

In order to solve the query ?- mortal(fred)., we used the rule we defined previously. This said that in order to prove someone mortal, we had to prove them to be human. Thus from the goal Prolog generates the subgoal of showing human(fred). Then Prolog matched human(fred) against the database. In Prolog we say that the subgoal succeeded, and as a result the overall goal succeeded. We know when this happens because Prolog prints “yes.”

Backtracking

Suppose that we have the following database:

```
eats(fred,oranges).
eats(fred,meat).
eats(fred,apples).
```

Suppose that we wish to answer the question *What are all the things that fred eats?*. To answer this we can use variables again. Thus we can type in the query:

```
?- eats(fred, Food).
```

Prolog will answer “Food=oranges”. At this point Prolog allows us to ask if there are other possible solutions. When we do so we get the following: “Food=meat”. Then, if we ask for another solution Prolog will give us: “Food=apples”.

If we ask for further solutions, Prolog will answer “no”, since there are only three ways to prove fred eats something. The mechanism for finding multiple solution is called backtracking. This is an essential mechanism in Prolog.

We can also have backtracking in rules. For example consider the following program.

```
likes(Person1,Person2):- hobby(Person1,Hobby), hobby(Person2,Hobby).
hobby(john,tennis).
hobby(tim,sailing).
hobby(helen,tennis).
hobby(simon,sailing).
```

If we now pose the query:

```
?- likes(X,Y).
```

Prolog will answer “X=john, Y=helen”. Then next solution that Prolog finds is “X=tim, Y=simon”.

Lists

Lists always start and end with square brackets, and the items they contain are separated by commas. Here is a simple list [a,simon,A_Variable,apple].

Prolog also has a special facility to split the first part of the list, called the head, away from the rest of the list, known as the tail. We can place a special symbol | (pronounced 'bar') in the list to distinguish between the first item in the list and the remaining list. For example, consider the following:

```
[first,second,third] = [A|B]
```

where A=first and B=[second,third]. The unification here succeeds. A is bound to the first item in the list, and B to the remaining list.

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 UNICEN. He has published over 15 papers in journals and conferences. The main contributions of his recent Ph.D. dissertation are reactive mobility by failure and MoviLog. His current research interests include development tools for mobile agents, intelligent agents, and logic programming. He is the chair of the VI Argentine Symposium on Artificial Intelligence (ASAI). More info can be found at http://www.exa.unicen.edu.ar/~azunino.

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 Universidad Nacional del Centro (UNICEN), Tandil, Argentina in 1988. Currently he is an Associate Professor at Computer Science Department and Head of the ISISTAN Research Institute of UNICEN. 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 conferences and journals about software engineering topics. His research interests include intelligent aided software engineering, software architectures and frameworks, agent technology and software visualization. More info can be found at http://www.exa.unicen.edu.ar/~mcampo.

Cristian Mateos is a MSc. candidate at the Universidad Nacional del Centro, working under the supervision of Marcelo Campo and Alejandro Zunino. He holds

a Systems Engineer degree from UNICEN. He has implemented part of the run-time support for reactive mobility by failure in MoviLog. He is investigating the relationships between Semantic Web Services and mobile agents using reactive mobility by failure.