MoviLog: A Platform for Prolog-Based Strong Mobile Agents on the WWW

Alejandro Zunino, Marcelo Campo, Cristian Mateos*

ISISTAN Research Institute - UNICEN University
Campus Universitario (B7001BBO)
Tandil, Bs. As., Argentina
{azunino; mcampo; cmateos}@exa.unicen.edu.ar

Abstract

Despite the wide availability of Java-based mobile agent platforms, mobile agent development is still recognized for being challenging and difficult. This is mainly caused by agents' location awareness, which implies that developers have to provide code for taking decisions about mobility, in addition to code implementing traditional stationary behavior. In this article we describe MoviLog, a mobile agent platform for building Prolog-based mobile agents called Brainlets. MoviLog implements a novel mobility mechanism, reactive mobility by failure (RMF), that is able to automatically migrate Brainlets based on its resource needs. MoviLog has been implemented as an extension of JavaLog, a multi-paradigm language that integrates Java and Prolog. The article also reports on experimental results on the usage of MoviLog and comparisons with other platforms.

Keywords: intelligent agents, mobile code, agent-oriented programming

1 Introduction

The huge amount of information available on the Internet became one of the main motivations for the development of mobile agent technology. A mobile agent is a computer program which represents a user in a computer network and is capable of migrating autonomously from host to host to perform some computation on behalf of the user [15, 14]. 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 services [7].

Intelligent agents have been traditionally considered as systems possessing several dimensions of attributes [10, 16, 6]. For example, [2] 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 it is possible to 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 a specific place of the network, and the second one is in charge of making decisions about mobility.

Clearly, mobile agents, as autonomous entities fully

*Also Comisión de Investigaciones Científicas (CIC)

aware of their location, have to be able to reason why, when and where to migrate in order to better use available network resources. Thus, in addition to stationary behavior, whose development is recognized for being challenging and highly complex [24, 8], 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 distributed systems and intelligent mobile agents, specially with respect to traditional non-mobile applications [12, 18].

Nowadays, after years of positive experiences using mobile agents in the academic world [14, 13], the question is no longer how advantageous the paradigm is, but how to fully exploit its advantages. Most mobile agent applications provide a *move* operation which is invoked when an agent wants to move to a remote site. Recent platforms support more elaborate abstractions that reduce the burden of programming mobile agents. For example, Concordia [23] and Ajanta [21] support itineraries and meetings among agents. Despite these advances, the agent developer is repeatedly faced with the three *www*-questions of mobile agents: *w*hy, *w*hen and *w*here to migrate.

This paper presents a new platform for mobile agents named MoviLog that uses stationary intelligent agents to assist the developer on managing mobility. MoviLog aims at reducing the development effort of mobile agents by automatizing decisions on why, when and where to migrate in order to better use available resources. MoviLog is an extension of the JavaLog framework [1, 25] which implements an extensible integration between Java and Prolog.

MoviLog provides mobility services enabling mobile logic-based agents, called Brainlets, to migrate between hosts using a strong mobility model. Besides extending Prolog with operators to implement proactive mobility, the main contribution of MoviLog is the incorporation of the notion of *reactive mobility by failure* (RMF). MRF acts when certain specially declared predicate fails, transparently moving the Brainlet to another host which has declared the same predicate. This means that MoviLog considers a set of web servers as a world-wide distributed database.

The article is structured as follows. The next section describes the main characteristics of the JavaLog framework. Section 3 introduces the MoviLog platform and examples of proactive and reactive mobility. Section 4 presents some experimental evaluations. Section 5 discusses the most relevant related work. Finally, in Section 6 concluding remarks are delineated and options for future work are discussed.

2 The JavaLog Framework

JavaLog is multi-paradigm language that integrates Java and Prolog implemented in Java [1]. The JavaLog support is based on an extensible Prolog interpreter designed as a framework. This means that the basic Prolog engine can be extended to accommodate different extensions, as multi-threading or modal logic operators, for example.

JavaLog defines the logic module as its basic concept of manipulation. In this sense, both objects and methods from the object-oriented paradigm are considered as modules. The elements manipulated by the logic paradigm are also mapped onto modules. Logic modules are defined by sets of Prolog clauses. JavaLog also provides two algebraic operators, union and overriding union, to combine logic modules.

Each agent encapsulates a complex 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 brain object also refers to logic modules sent by the object-agent for working on them from that instant.

2.1 An Example

The following example involves customer agents capable of selecting and buying different articles based on users' preferences. A *CustomerAgent* class defines the behavior of customers whose preferences are expressed through a logic module received as a

parameter. The *CustomerAgent* class is implemented in the following way:

```
public class CustomerAgent {

 private PlLogicModule pref;

 public CustomerAgent(PlLogicModule pref) {
  this.pref = pref;
 }

 public boolean buyArticle(Article art) {
  pref.enable(); type = art.type;
  ...
  if(?-preference((#art#,[#type#,#brand#,
   #model#,#price#]).)
    buy(art); pref.disable();
 }

 ...
}
```

The example defines a variable named *pref*, which references a logic module including user preferences. When the agent needs to decide whether to buy a given article, user preferences are analyzed. The *buyArticle* method first enables the *pref* logic module to be queried. In this way, the knowledge included in that module is added to the agent knowledge. Then, an embedded Prolog query is used to test whether to buy the article. To evaluate *?-preference(#art#, [#type#, #brand#, #model#, #price#])*, *pref* clauses are used. The query contains a Java variable enclosed into #. This mark allows the programmer to use Java objects inside Prolog clauses. In addition, *send* can be used to send a message to a Java object from a Prolog program. For instance, *send(#art#, brand, Brand)* in Prolog is equivalent to *Brand = art.brand()* in Java. Finally, the *buyArticle* method disables the *pref* logic module. This operation deletes the logic module from the active database of the agent.

In order to create a customer agent, a logic module with the user preferences have to be provided, enclosed between {{ and }} as follows:

```
CustomerAgent anAgent =
 new CustomerAgent( {{
  preference(car,[ford, Model, Price]) :-
   Model > 1998, Price < 200000.
  preference(motorcycle,[yamaha, Model,
   Price]) :- Model >= 1998, Price < 9000.
 }});
```

3 The MoviLog Platform

MoviLog is, essentially, an extension of the JavaLog framework to support mobile agents on the web. MoviLog implements a strong mobility model for a special type of logic modules, 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 origin host.

In order to enable mobility across web sites, each web server belonging to a MoviLog network must be extended with a MARlet (Mobile Agent Resource). A MARlet extends the Java servlets support by 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 performing logic queries. In this sense, a MARlet can also be used to provide inferential services to agents as well as legacy web applications.

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. This mechanism is supported by a number of stationary agents distributed across the network called Protocol Name Servers (PNS). These agents provide 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

The *moveTo* built-in predicate provided by MoviLog permits a Brainlet to autonomously migrate to another host. Before migration, MoviLog in the local host serializes the Brainlet and its 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. Upon receipt of an agent, the MoviLog platform in the remote host reconstructs

the Brainlet and the objects it refers to, and then it resumes the execution of the agent. Eventually, after performing some computations, 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 a user's preferences. The *buy* clause looks for offers available in a number of sites, 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 and so on.

```
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, 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)), 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. That is, mobile Prolog programs cannot necessarily be built in the declarative way as a normal Prolog program is, forcing to implement 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*.

3.2 Reactive Mobility by Failure

The MoviLog platform provides a new mechanism for mobility called *reactive mobility by failure* (RMF) which aims at reducing the effort of developing mobile agents by automatizing some decisions about mobility. RMF is based on the assumption that mobility is orthogonal to the rest of attributes that an agent may possess (intelligence, agency, etc) [2]. Under this assumption it is possible to think of a separation between these two functionalities or concerns at the implementation level [5]. RFM 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, as depicted in Fig 1.

RMF is a mechanism that, when a certain predicate fails, transparently moves a Brainlet to another site having definitions for such a predicate and continues the normal execution trying to find a solution. The implementation of this mechanism requires the MoviLog inference engine to know where to send the Brainlet. For this, MoviLog extends the normal definition of a logic module with *protocol sections*, which define predicates that can be shared across the network.

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, the MoviLog engine 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 MARlets offering 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

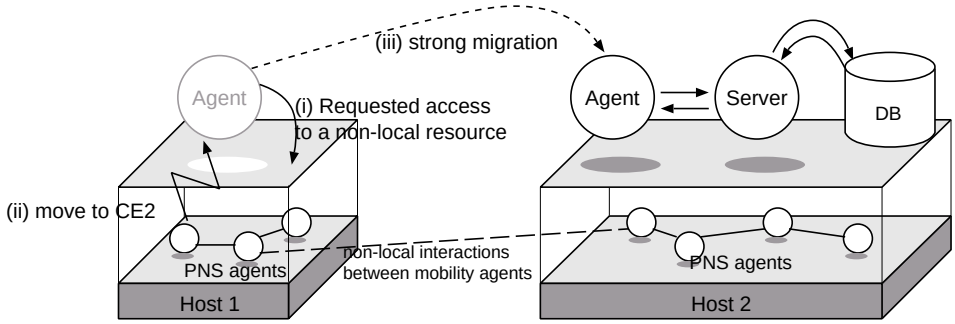


Figure 1: Reactive Mobility by Failure

of the MARlets 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 MARlet fails to probe *g*, it continues with the next MARlet in D_R. When no more possibilities are available, the Brainlet is moved to its origin.

The following code shows the implementation of the customer agent combining both mobility mechanisms. As can be noted, the solution using RMF looks much like a regular Prolog program. 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 MARlets offering the same protocol before returning to the origin 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 MARlet offering that article in order to buy it. Certainly, this solution is simpler than the one using just proactive mobility.

```
PROTOCOLS
  article(A,Offer,Email).

CLAUSES
  preference(car, [ford, Model, Price]) :-
    Model > 1998, Price < 20000.
  preference(tv,[sony, Model, Price]) :-
    Model = 21in, Price < 1000.

  lookForOffers(A, [O|RO], [O|Roff]) :-
    article( A, Offer, Email),
    thisSite(ThisSite),
    assert( offer(ThisSite, Offer, Email) ),
    fail.
  lookForOffers(A, _, Offers) :- !,
    findall(_, offer(S,O,E), Offers).
  buy(Art) :- lookForOffers(Art,R,Offers),
    selectBest(Offers, offer(S,O,E)),
    moveTo(S), buy_article(O, E),
    return.
  ...
  ?- buy(Art).
```

It is worth noting that in the example the articles offered by all the sites with a protocol article are collected through RMF (when *article* fails at the current MARlet the Brainlet is transparently moved).

3.2.1 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.

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 $\langle t_1, t_2, \ldots, t_n, t_i, V = \{\}, \langle H_t = \langle\,\rangle, H_v = \langle\,\rangle, PH_t = \langle\,\rangle\rangle\rangle$
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 not such an 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 its origin.
16: **end if**

3.2.2 Distributed Backtracking and Consistency Issues

Mobile Prolog, and particularly, the RMF mobility model generates several tradeoffs related 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 PNS 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 consists on updating the local view of a Brainlet 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 [11] (a Java-based framework for mobile

agents) and Jinni [20] (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, edition, author, subject, etc.

The implementation of the application with MoviLog using RMF was straightforward (only 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. As a result the total size of the application was 22605 lines of Java source code. 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. It is worth noting that MoviLog provides powerful abstractions for rapidly developing intelligent and mobile agents. On the other hand, the others platforms are more general, thus 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 1600 KB. 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 (1600 KB, 11135 books approx.). We ran each test case 5 times and measured the running time. Fig. 2(b) 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 (1600 KB, 11135 books approx.) distributed across three hosts. Fig. 2(a) shows the network traffic measured in packets versus the number of books that matched the user's preferences.

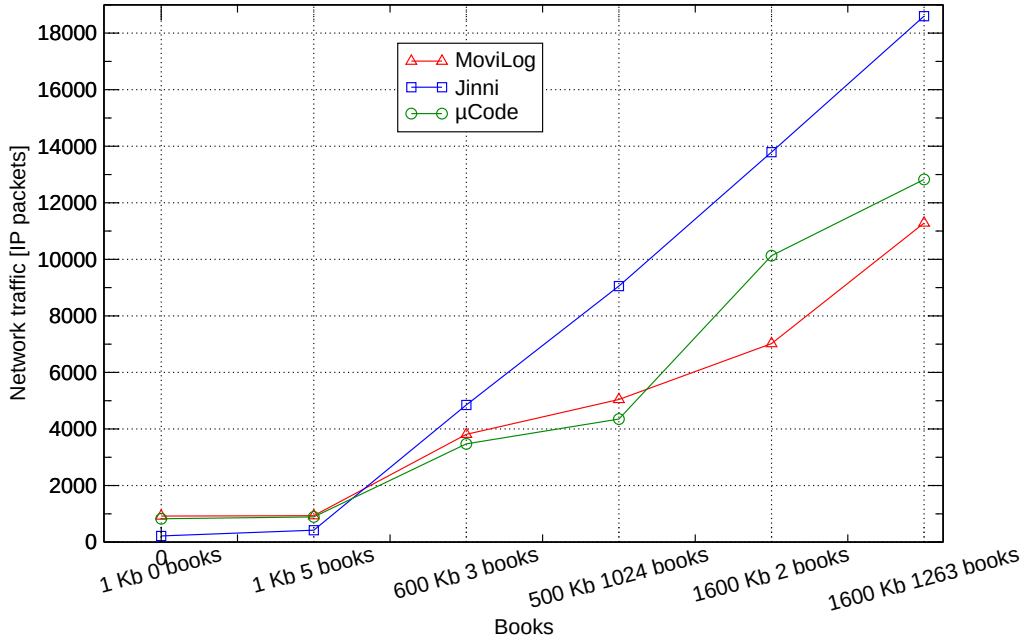
From the figures we can conclude that MoviLog and its RMF do not affect negatively neither the performance nor the network traffic, while considerably reducing the development effort. The next section discusses previous work related to MoviLog.

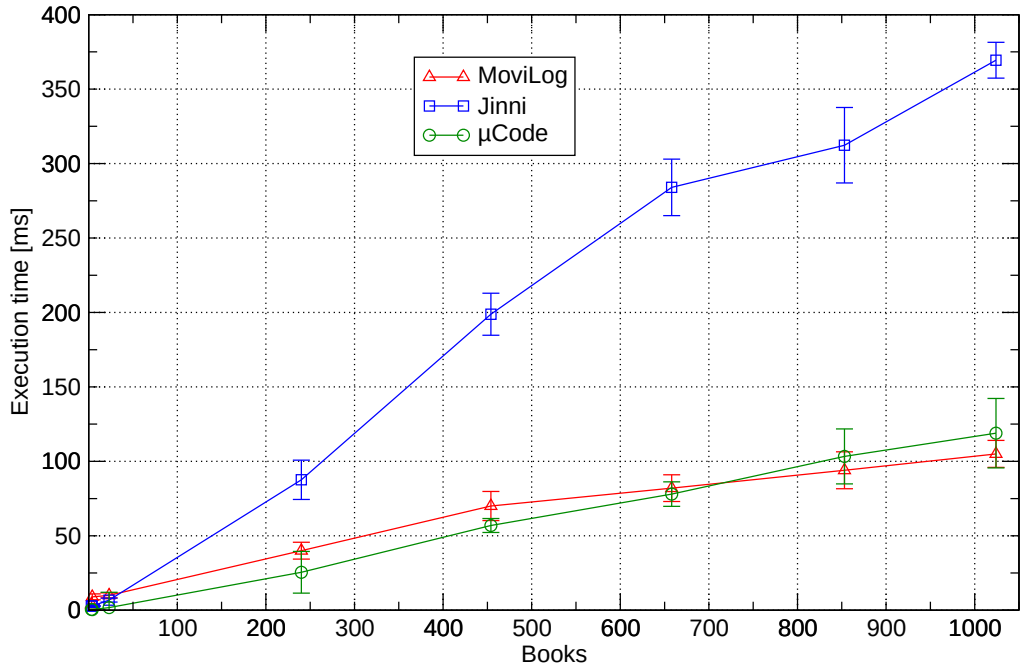
5 Related Work

At present, Java is the most commonly used language for the development of mobile agent applications. Aglets [9], μCode [11], Concordia [23], AgentSpace [17] and Ajanta [21] are examples of Java-based mobile agent systems. These systems provide a weak mobility model, forcing a less elegant and more difficult to maintain programming style [18]. Recent works such as NOMADS [19] and WASP [4] 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, namely its ubiquity, portability and compatibility across different platforms and operating systems.

The logic-programming paradigm represents an appropriate alternative to manage agents mental attitudes. Ciao Prolog [3], Jinni [20] and Mozart/Oz [22] are some proposed logic programming-based languages for multi-agent systems development. Ciao Prolog is a logic programming system that generates multi-platform executables based on compilation to bytecode allowing code mobility across platforms. Ciao Prolog allows goals to be launched in separate threads. These threads can be communicated and synchronized by using the shared Prolog database. It also defines a special kind of object, called active, which can be used to implement distributed knowledge bases in which updates to global facts are automatically available to remote agents.

Jinni (Java INference engine and Networked Interactor) [20] is based on a limited subset of Prolog. Jinni supports strong mobility. However, the language lacks adequate support for mobile agents since its notion of code and data closure is limited





(b) Running Time

Figure 2: Performance Comparisons

to the currently executing goal. As a consequence, programmers using Jinni have to manage code and data closure by programming mechanisms for saving and restoring an agent's code and data.

Mozart [22] 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. Despite this shortcoming, Mozart offers a clean and easy syntax for developing distributed applications with little effort.

The main difference between MoviLog and other platforms is RMF, which reduces development effort by automatizing some decisions about mobility, and its multi-paradigm syntax, which provides mechanisms for developing intelligent agents with knowledge representation and reasoning capabilities. MoviLog reduces and simplifies the effort of mobile agent development, while being as fast as any Java-based platform.

6 Conclusions

This paper presented MoviLog, a multi-paradigm language based on Prolog and Java, that supports reactive mobility by failure. RMF simplifies mobile agent development by automatizing decisions about mobility. At the same time, mobile agents are able to autonomously migrate by using proactive strong mobility mechanisms. The result is a platform that combines both reactive and proactive migration to cope with the complexities of mobile agent development.

We are working on defining the formal semantics of the language and its relationship with traditional Prolog semantics. Another line of research is the integration of MoviLog with Web services and the Semantic Web.

References

[1] Analía Amandi, Alejandro Zunino, and Ramiro Iturregui. Multi-paradigm languages supporting multi-agent development. In Francisco J. Garijo and Magnus Boman, editors, *Multi-Agent System Engineering*, volume 1647 of *Lecture Notes in Artificial Intelligence*, pages 128–139, Valencia, Spain, June 1999. Springer-Verlag.

[2] Jeffrey M. Bradshaw. *Software Agents*. AAAI Press, Menlo Park, USA, 1997.

[3] Daniel Cabeza, Manuel Hermenegildo, and Sacha Varma. The PiLLoW/CIAO Library for Internet/WWW Programming using Computational Logic Systems. In P. Tarau, A. Davison, K. DeBosschere, and M. Hermenegildo, editors, *Proc. 1st Workshop on Logic Programming Tools for INTERNET Applications*, 1996.

[4] Stefan Fünfrocken and Friedemann Mattern. 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, 1999.

[5] Alessandro Garcia, Christina Chavez, Otavio Silva, Viviane Silva, and Carlos Lucena. 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.

[6] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 1994.

[7] Robert S. Gray, George Cybenko, David Kotz, and Daniela Rus. Mobile agents: Motivations and state of the art. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2001.

[8] Nicholas Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[9] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Mobile Agents with Java Aglets*. Addison-Wesley, Reading, MA, USA, September 1998.

[10] Hyacinth Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244, September 1996.

[11] Gian Pietro Picco. μCode: A Lightweight and Flexible Mobile Code Toolkit. In K. Rothermel and F. Hohl, 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, 1998.

[12] Gian Pietro Picco, Antonio Carzaniga, and Giovanni Vigna. Designing distributed applications with mobile code paradigms. In R. Taylor, editor, *Proceedings of the 19th International Conference on Software Engineering*, pages 22–32. ACM Press, 1997.

[13] Kurt Rothermel and Fritz Hohl, editors. *Second International Workshop on Mobile Agents (MA'98)*, volume 1477 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Germany, 1998.

[14] Kurt Rothermel and Radu Popescu-Zeletin, editors. *First International Workshop on Mobile Agents (MA'97)*, volume 1219 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Germany, 1997.

[15] Daniela Rus, Robert S. Gray, and David Kotz. Transportable information agents. In *International Conference on Autonomous Agents*, pages 228–236, February 1997.

[16] Yoav Shoham. An overview of agent-oriented programming. In Jeffrey M. Bradshaw, editor, *Software Agents*, chapter 13, pages 271–290. AAAI Press / The MIT Press, 1997.

[17] Alberto Silva, Miguel Mira da Silva, and José Delgado. An Overview of AgentSpace: A Next generation Mobile Agent System. In K. Rothermel and F. Hohl, editors, *Proceedings of the* 2^{nd} *International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 148–159. Springer-Verlag: Heidelberg, Germany, 1998.

[18] Alberto Silva, Artur Romao, Dwight Deugo, and Miguel Mira da Silva. Towards a Reference Model for Surveying Mobile Agent Systems. *Autonomous Agents and Multi-Agent Systems*, 4(3):187–231, September 2001.

[19] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, Renia Jeffers, and Timothy S. Mitrovich. An Overview of the NOMADS Mobile Agent System. In 6^{th} *ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages*, 6 2000.

[20] Paul Tarau. Jinni: a lightweight java-based logic engine for internet programming. In Kostis Sagonas, editor, *Proceedings of JICSLP'98 Implementation of LP languages Workshop*, Manchester, U.K., June 1998. invited talk.

[21] Anand R. Tripathi, Neeran M. Karnik, Tanvir Ahmed, Ram D. Singh, Arvind Prakash, Vineet Kakani, Manish K. Vora, and Mukta Pathak. Design of the Ajanta System for Mobile Agent Programming. *Journal of Systems and Software*, 2002. to appear.

[22] Peter Van Roy and Seif Haridi. 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).

[23] David Wong, Noemi Paciorek, Tom Walsh, Joe DiCelie, Mike Young, and Bill Peet. Concordia: An infrastructure for collaborating mobile agents. In Rothermel and Popescu-Zeletin [14], pages 86–97.

[24] Michael Wooldridge and Nicholas Jennings. Pitfalls of agent-oriented development. In Katia P. Sycara and Michael Wooldridge, editors, *Proceedings of the 2nd International Conference on Autonomous Agents (AGENTS-98)*, pages 385–391, New York, May 9–13 1998. ACM Press.

[25] Alejandro Zunino, Luis Berdún, and Analía Amandi. Javalog: un lenguaje para la programación de agentes. *Revista Iberoamericana de Inteligencia Artificial*, 13:94–99, 2001.