This is a preprint of the article: "Extending MoviLog for Supporting Web Services. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley. Vol 33, Num. 1. pp 11-31. 2007. doi:10.1016/j.cl.2006.02.001"

The final publication is available at http://dx.doi.org/10.1016/j.cl.2006.02.001

Extending MoviLog for Supporting Web Services

Cristian Mateos [1] Alejandro Zunino [*,1] Marcelo Campo [1]

ISISTAN Research Institute. UNICEN University. UNCPBA. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (2293) 440363. Fax.: +54 (2293) 440363

Abstract

Web Services enable computers to interact and exploit Web-accessible programs without human intervention. Despite researchers agree that mobile agent technology will obtain significant benefits from this line of research, the lack of proper development tools hinder the widespread adoption of mobile agent technology on the Web. This paper describes a novel programming language called WS-Log whose goal is to provide a tight integration between mobile agents and Web Services. Examples and experimental results showing some of the advantages of WS-Log are also reported.

Key words: Mobile agents, Logic Programming, Web Services, Intelligent Agents

1 Introduction

The development of the Web started 16 years ago as an ambitious project proposal [1]. The first concrete result of this project was the World Wide Web, an abstract information space with the largest collection of documents that has ever existed [2]. The WWW quickly became popular among developers because it hid the diversity of software, hardware and network architectures existing then. This information space was designed to be fully distributed and without a central control or coordination. The mechanism for browsing Web content was mainly designed for human use [3]: a user consults and interprets

* Corresponding Author.
Email addresses: cmateos@exa.unicen.edu.ar (Cristian Mateos), azunino@exa.unicen.edu.ar (Alejandro Zunino).
[1] Also Comisión Nacional de Investigaciones Científicas y Técnicas (CONICET)

documents by reading HTML (Hyper Text Markup Language) pages that are rendered by a special application called *Web browser*.

Few years ago, the WWW started to evolve into a global network of *Web Services* [4] whose goal is to achieve automatic interoperability between applications and Web resources. A Web Service [5,6,7] is any Web-accessible program or resource. Web Services can be seen as a set of programs interacting through the WWW without human intervention. In this sense, the Web that is coming is not only about information sharing, but also about providing programs, including intelligent agents, with an infrastructure to use Web-accessible resources [8].

Recently, various applications that interoperate through Web Services have been developed, at first for B2B (Business-to-Business) and e-commerce applications. For example, many popular Web sites such as Amazon [2], eBay [3] and Google [4] offer Web Services for applications that expose the same information and functionality a user can access by using a regular Web browser. However, there is an increasing need for automating the way programs interact with Web information and services. Until now, this interoperation has been handled by using hand-coded programs that interact with Web-accessible services to get and then parse their answer, usually HTML data. This approach is weak since it depends on the format of the HTML content and the interfaces for accessing those services.

Many researchers agree that *mobile agents* will have a fundamental role in the future of the Web [9,10]. A mobile agent is a computer program that represents a user in a network, and is able to migrate from site to site to perform tasks for the user [11]. Mobile agents have several properties that make them suitable for exploiting the potential of the Web. Some of the most significant advantages of mobile agents are their support for disconnected operations, heterogeneous systems integration, robustness and fault-tolerance [12].

Despite the number of advantages mobile agents offer [13], this technology has shown difficulties when used for interacting with Web content [9]. Developers are usually forced to pay attention to low-level implementation details for invoking Web services (setting up connections, formatting data, converting data-types, etc.) and handling complex migratory code, rather than focusing on agents' behavior. Undoubtedly, this fact represents a hurdle for integrating mobile agents with Web Services. In this sense, we believe there is a need for a mobile agent development tool for solving these problems and, at the same time, preserving the key benefits of mobile agent technology for building massively distributed applications.

[2] Amazon Bookstore: http://www.amazon.com/gp/aws/landing.html
[3] eBay: http://developer.ebay.com/DevProgram/index.asp
[4] Google Search Engine: http://www.google.com/apis/

WS-Log is a platform for building and deploying Prolog-based intelligent mobile agents for the WWW based on MoviLog [14,15]. WS-Log supports a novel mechanism for handling resource access named RMF (Reactive Mobility by Failure), which allows programmers to easily build mobile agents that use Web Services without worrying about services location or invocation details.

This article is structured as follows. The next section discusses the technological backbone of Web Services. Section 3 presents the most relevant related work. Section 4 introduces the WS-Log programming language, focusing on its syntax and runtime support. Section 5 briefly discusses the design and implementation of WS-Log. Section 6 presents a sample application. Section 7 reports some experimental results. In section 8 concluding remarks and future works are presented. Finally, appendix 8 introduces Prolog.

2 Web Services

Web Services are a suitable model to allow systematic interactions of Web applications and integration of legacy platforms and environments. Web Services model mostly relies on technologies based on XML (Extended Markup Language) [16], a structured language that extends and formalizes HTML. In this sense, the W3C Consortium has developed SOAP (Simple Object Access Protocol) [17], a communication protocol based on XML. Nowadays, SOAP is widely accepted and is included in most of the communication infrastructure proposed for integrating applications and Web Services. In addition, languages for describing Web Services have been developed. The most notorious example is WSDL (Web Service Description Language) [18], an XML-based language which allows developers to create service descriptions as a set of operations over SOAP messages. From a WSDL specification, a program can discover the specific services a Web site provides, and how to use and invoke these services.

As a complement to WSDL, UDDI (Universal Description, Discovery and Integration) [19] has been proposed. UDDI provides mechanisms for searching and publishing services written preferably in WSDL. With UDDI, Web Service providers such as enterprises or organizations register information about the services they offer, thus making this information available to potential clients. The information stored into UDDI registries ranges from WSDL files describing services to useful data (e-mail, Web pages, etc.) for contacting the associated provider.

The most widely accepted conceptual architecture for Web Services is shown in figure 1. Here, a Web Service is defined as an interface describing a collection of operations that are network accessible through standardized XML messaging. WSDL is used to describe the software interface to the Web Service, and all

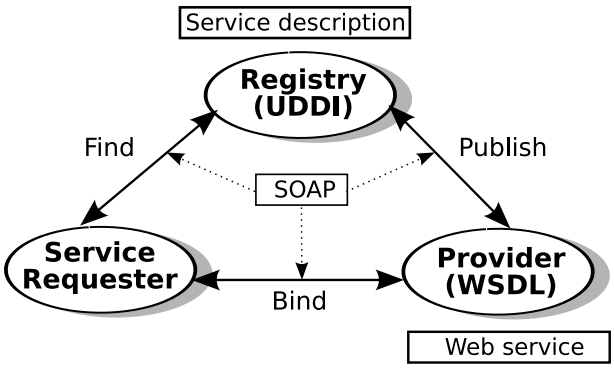

Figure 1. Web Service conceptual architecture[20]

interactions between any pair of components are supported through SOAP.

The architecture encompasses three classes of components: service providers, service requesters and service registries. A service provider creates a WSDL document describing its Web Service and publishes this document to a service registry such as UDDI. A service requester can use a registry to find a Web Service that matches its needs and retrieve the corresponding WSDL document. Using the information provided by a WSDL document, a service requester invokes the operations of the provider's Web Service.

The next section surveys some tools for agent development with support for Web Services.

3 Related Work

Web Services have motivated new research and development on agent tools. Some of the most relevant of these tools are ConGolog [21], IG-JADE-PKSLib [22], MWS [23] and Jinni [24].

ConGolog is an extension of Golog [25], a logic-based high-level programming language designed to specify and execute complex plans in dynamic domains. ConGolog lets programmers to create plan tasks using Web Services described in DAML. DAML (Darpa Agent Markup Language) [26] is a language for Web Services metadata annotation belonging to the family of languages proposed for the Semantic Web [27]. Despite its benefits for automatic manipulation of service metadata, the language has some limitations when working with services that depend on each other. A programmer is not able to specify, for example, the need to execute a login Web Service before executing a query service. In addition, ConGolog does not support agent mobility.

IG-JADE-PKSLib is a toolkit for programming cognitive agents focused on the problem of Web Service Composition (WSC). WSC can be defined as follows:

given a set of Web Services and a generic (usually user-defined) goal, find a valid composition of those services to carry out the goal. To cope with this, IG-JADE-PKSLib uses PKS (Planning with Knowledge and Sensing) [28], a planning algorithm based on a generalization of STRIPS [29], producing pre-compiled plans which invoke Web Services when they are executed. From the experiments reported in section 7 we noted that this language is not suitable for on-the-fly composition and execution of Web Services, mainly because of its poor performance. Besides, agent mobility is not supported.

MWS (Mobile Web Services) is a generic framework for programming and deploying Web Services implemented as mobile agents. A *mobile Web Service* is composed of descriptions of interaction processes with other services, resources required to execute, and migration policies. In this way, MWS enables users to implement a wide range of mobile services by combining different settings of the three mentioned elements. Unfortunately, MWS lacks support for common agent requirements such as knowledge representation, reasoning and high-level communication.

Jinni (Java INference engine and Networked Interactor) [24] is a Prolog-based interpreted language for programming applications in heterogeneous networks. Jinni supports code migration only by sending a Prolog goal to another Jinni server for remote execution. This approach limits the usefulness of mobility and makes agent development hard. Interaction with Web Services is supported through *wrappers*, which abstract the necessary low-level details (endpoints, protocols, etc.) to contact a service. The drawback of this approach is that programmers have to provide a new wrapper for each service they require. Usually, these wrappers include specific low-level access code that is not only difficult to implement, but also hard to debug and maintain.

All in all, despite some interesting advances towards the integration of agents and Web Services have been made, current approaches have the following problems: bad performance/scalability (IG-JADE-PKSLib), no/limited mobility (IG-JADE-PKSLib, ConGolog, Jinni) and lack of support for agents (MWS).

The next section describes WS-Log, a new language for programming mobile agents. WS-Log aims at solving the problems mentioned before by allowing the development of mobile agents which interact with Web Services.

4 WS-Log

WS-Log is a language for programming Prolog-based mobile agents in the WWW. WS-Log is based on MoviLog [14], a platform for building intelligent

mobile agents using a strong mobility model [30], where agents execution state is transferred transparently on migration. MoviLog is built as an extension of JavaLog [31], a framework for agent-oriented programming based on Java and Prolog.

Besides providing basic strong mobility primitives the most interesting aspect of WS-Log is the notion of Reactive Mobility by Failure (RMF) [15], mechanism not exploited by any other tool for mobile agents. Conceptually, a failure is defined as the impossibility of an executing agent to use some needed resource at the current site [15].

Before going into details on the execution model of WS-Log we will briefly explain the composition of each agent. Agents in WS-Log are called *Brainlets*. Each Brainlet has Prolog code (appendix 8 introduces Prolog) that is organized in two sections: *predicates* and *protocols*. The first section defines the agent behavior and data. The second section declares rules that are used by the RMF for managing mobility.Sintactically, the code of a Brainlet has the following form:

```
PROTOCOLS
  % Prolog facts representing protocols
CLAUSES
  % Prolog rules implementing agent behaviour
```

The RMF states that when a predicate declared in the protocols section of an agent fails, WS-Log moves the Brainlet and its execution state to another site that contains definitions for the predicate and then resumes the Brainlet execution. Indeed, not all failures trigger mobility, but only failures caused by predicates declared in the protocols section. The idea is that normal predicates are evaluated with the regular Prolog semantics, but predicates for which a protocol exists are treated by the RMF so that their failure may cause migration. To distinguish between Prolog failures with the traditional semantics and failures handled by the RMF we will refer to the latter as m-failures.

The next example presents a simple Brainlet whose goal is to first collect temperature values from different sites, and then calculate the average of these values. Each measurement point is represented by a WS-Log site with a special process that stores, at a regular basis, the last measurement T in the site local database as a *temperature(T)* predicate. The code implementing the Brainlet is:

```
PROTOCOLS
  protocol(temperature,[arity(1)]).
CLAUSES
  average(List, Avg):-...
  getTemp(Curr, List):- temperature(T), thisSite(S),
```

```
                        M = measure(T,S),
                        not(member(M, Curr)),
                        getTemp([M|Curr], List).
getTemp(Curr, Curr).
average(Avg):- getTemp([], List), average(List, Avg).
?-average(Avg).
```

The idea of the program is to force the Brainlet to visit all the sites, getting at each one the current temperature, and then computing the average of those values. The potential activation point of the RMF is the *temperature(T)* predicate. PROTOCOLS declares that the evaluation of *temperature(T)* must be handled by the RMF. As a result, if the evaluation of this predicate fails at a site *S,* the RMF will transfer the Brainlet to a site containing definitions for *temperature/1* (predicates with functor "temperature" with one argument). The evaluation of *getTemp* will end successfully once all the sites offering *temperature/1* have been visited.

In order to explain the execution of the program we will consider a network composed of three WS-Log enabled sites. The idea is to trigger mobility upon *m*-failures of predicates *temperature/1* and hence forcing the Brainlet to visit the three sites S_1, S_2 and S_3. Let us suppose that we launch the program from S_1, by invoking *?-average(Avg)*. The code behaves the same as a regular Prolog program up to the point when *getTemp* tries to evaluate *temperature* for the second time. In this case, the evaluation of *temperature* will fail because the temperature value stored at S_1 has been already collected. Considering that *temperature* has been declared as a protocol, an *m*-failure will occur. As a consequence, the RMF run-time will search for sites providing clauses *temperature/1* to migrate the agent and to try to reevaluate the goal there. Note that there are two options, either S_2 or S_3. Let us assume that the RMF selects S_2. Then, after the migration of the agent to S_2, *getTemp* will collect the local temperature until no more options are available. At this point an *m*-failure will occur and the RMF will select M_3. After evaluating once at M_3, *temperature* will fail again. In this case there will be no more options left for migrating the agent. Then, it will be returned [5] to its origin (S_1) by the site S_3. Finally, the result of *?-average(Avg)* will be the average of the values [temperature(T_3), temperature(T_2), temperature(T_1)].

In the example, the agent visits all the sites containing temperature values. This behaviour is not forced by WS-Log, but by *getTemp*, because it evaluates all the temperature predicates in order to make *not(member(M, Curr))* true. In other words, when an *m*-failure occurs, the RMF moves the agent to one

[5] After a successful evaluation of a predicate that *m*-failed an agent does not return to its origin. It returns if it finishes its execution or fails (no more alternatives are available).

site only, leaving remaining options as *backtracking* points.

It is worth noting that WS-Log does not restrict the programmer to use the RMF for handling mobility. Instead, by declaring protocols the programmer is able to select which predicates of a Brainlet code can trigger mobility. At an extreme, a Brainlet may not declare any protocol. This does not imply that mobility is not available, but mobility is in charge of the programmer as in most tools and languages for mobile agents. Indeed, protocol declarations allow the programmer to use the RMF and traditional proactive mobility in the same Brainlet, depending on his requirements.

The first version of the RMF [14] was designed to automate mobility decisions such as when and where to move a Brainlet, there are situations where performance may be bad. For example, if the size of a Brainlet is greater than the size of a requested resource. Clearly, it is convenient to transfer a copy of the resource from the remote site, instead of moving the Brainlet to that site. If the requested resource is a Web Service, the proper way to use it is to invoke the service by using SOAP, thus transferring only input arguments and results. Finally, the interaction of an agent with a large database can be better done by moving the agent to the remote site, and then locally interacting with the database. In this case, database access by copy is unacceptable because it might use too much network bandwidth.

WS-Log improves the RMF by including extra methods for accessing resources besides agent mobility. In this sense, proper methods for interacting with Web resources have been developed, such as remote invocation, for the case of Web Services, and replication or copy of resources between sites, for the case of Web pages, data and code. In addition, WS-Log extends the RMF by providing decision mechanisms for selecting an access method based on both the resource type and environmental conditions, such as the total free memory available at a given site, the network transfer rate, or any user provided metric. Table 1 shows some examples of resources and their corresponding valid access methods. In the table an *external* resource refers to a resource hosted at a site that does not support Brainlets (most public Web servers).

When an m-failure occurs, there may be more than one site offering the required resource. WS-Log is able to decide an ordering for accessing these sites if a Brainlet requires visiting more than one of them. In addition, since depending on resource type several access methods may be suitable, WS-Log can apply different tactics to select the most convenient one. Both, ordering sites and choosing an access method are decided by WS-Log through *policies*. Policies are decision mechanisms based on system metrics such as network traffic, distance between sites, CPU load, etc. For example, one may specify that any access to a certain large database should be done by moving the agent where the database is located, rather than performing a time and bandwidth-

Table 1
Examples of resources and valid access methods

Resource	**Access methods**
Web Service	move or remote invocation
Prolog code	copy, move or remote invocation
Public file or Web page	copy or move
Large database or private file	move
External Web Service	remote invocation
External Prolog code	copy or remote invocation
External file or Web page	copy

consuming copy operation of the data from the remote site. Besides, WS-Log lets the programmer to define custom policies for adapting the RMF to his requirements.

The next subsection describes further details on the RMF in WS-Log. Then subsection 4.2 explains the policy programming support offered by WS-Log.

4.1 Reactive Mobility by Failure in WS-Log

As mentioned earlier, protocols are resource descriptors or, in other words, logical pointers a Brainlet uses to reference the set of resources it could need along its lifetime. Protocols are declared in the PROTOCOLS section of a Brainlet code as follows:

protocol(resourceKind ,[prop$_1$,prop$_2$,...,prop$_n$], accessPolicy)

where:

- *resourceKind* is a literal (atom) representing the general resource category which the resources referenced by the protocol belong to. Basically, a category stands for any kind of resources accessible to Brainlets, such as files, databases, Prolog clauses, code libraries, Web Services, etc.
- The second argument of the Prolog structure corresponds to the list of properties the desired instances must match. This list permits a Brainlet to reference different subsets from the set of resources belonging to the *resourceKind* class. Each property has the format *A(B)*, where A is the property name, and *B* contains the property value.
- Finally, *accessPolicy* contains the identifier of the policy used by the agent to choose an unique instance of the resource when more than one are available, and to select the access method (remaining instances are left as *backtracking*

points). This policy must be declared in the POLICIES section, as will be explained later. A value of "none" indicates that all access decisions over the referenced set of resources are delegated to the RMF.

For example, a protocol for a Web Service resource includes the operation name [6], its input arguments, and its output(s). The following code declares a protocol describing a Web Service for logging on a server that requires validation through a user name and password mechanism:

```
protocol(web-service,
         [operation-name(doLogin),
          input([user(U),pass(P)]),
          output(X)],none)
```

Roughly, the previous declaration enables the RMF mechanism to act whenever a login service can not be found at the local site. When this occurs, the RMF searches for published Web Services including an operation whose name literally matches "doLogin". Also, the search is constrained to those operations whose input is composed of two arguments (user and password), but does not further restrict the operation outputs (could be a *boolean* or a *string* indicating the success/failure of the login attempt). Moreover, since no access policy has been specified for the protocol, the RMF run-time is in charge of selecting an appropriate method for contacting services. Note that a different protocol for logging a server with a specific user (and perhaps an user-defined access policy) could be declared, just by replacing the variable U with the desired username.

As the reader can see, some attributes of the Web Service such as the server address or the transfer protocol are not specified. This information is encapsulated by each login service instance in its WSDL file, and is extracted and used at runtime when a specific instance is selected. These kinds of attributes are the *hidden* properties of a resource, that is, information only accessible to the WS-Log platform and therefore not taking part in the protocol matching process. Hidden properties and *public* properties (operationname, input and output in our example) must be supplied by providers when they publish a service or a resource.

Another example is presented next. It consists of a mobile agent for distributed searching of text files. In particular, the Brainlet has to find the files whose name is "ContactList.txt" containing the string "John Smith". For simplicity, wild cards are not considered. The code that implements the agent is:

```
PROTOCOLS
```

[6] This property is mandatory, since within a WSDL definition a single Web Service may be composed of more than one operation

```
  protocol(file, [name(X)], none).
POLICIES
  % empty section
CLAUSES
  searchForFiles(Text, FileName, Files):-
    assert(filesFound([])),
    findFiles(Text, FileName),
    filesFound(Files).
  findFiles(Text, FileName):-
    file([name(FileName)], FileProxy),
    analizeFile(Text, FileProxy), !, fail.
  findFiles(_, _).
  analizeFile(Text, FileProxy):-
    searchText(Text, FileProxy),
    send(FileProxy, 'getURL', [], FileURL),
    addNewFile(FileURL).
  addNewFile(FileURL):-
    filesFound(TempFiles),
    retract(filesFound(_)),
    assert(filesFound([FileURL|TempFiles])).
  searchText(Text, FileProxy):- ...
  ?-searchForFiles("ContactList.txt","John␣Smith", Result).
```

In this case, the agent defines only one protocol that declares the need for accessing, at some point of its execution, one or more instances of a resource of type *file*. Also, the protocol indicates that those instances must have their name attribute as a public property. In other words, every time the agent evaluates a rule requiring a file, the RMF will handle the request. In the example, the programmer does not specify any rule for handling the retrieval of the file. However, it could be convenient to select the access method according to the file size. For example, if it exceeds a certain size, the RMF may decide to migrate to the site where the file is located rather than transferring the file, because this latter approach might waste network bandwidth. In the next subsection we will show how to program these decisions by using resource access policies programming support provided by WS-Log.

The Brainlet begins its execution by evaluating *?-searchForFiles*. When a predicate fails accessing a *file*, WS-Log asks the current site for the list of remote resource instances matching the protocol requested. To be more exact, WS-Log searches for those instances which have been published under the "file" resource category with a public property "name". As the agent does not declare any policy for accessing the file, WS-Log chooses any instance from the list and a proper access method, and then accesses the file. When the *file* predicate is reevaluated, WS-Log uses another element of the list as the file protocol being processed. Once all elements have been consumed, the predicate cannot be further reevaluated, and the file searching process ends. After this, the agent

returns to the site from where it was launched.

The access to each file instance is requested through the predicate *file([name(FileName),FileProxy)*, which also filters those files whose name is not the same as *FileName*. As a result, the *FileProxy* variable is instantiated with an object which hides the real location of the file, and provides a handler to read its contents. From a WS-Log program, it is possible to call a Java method by using the predicate *send(Instance, Method, Args, Result)*. In the example, the proxy *getURL()* method is called to obtain the real location of the file being processed.

4.2 Resource access policies

Programmers can customize the way agents interact with resources to fit specific applications constraints and needs. WS-Log provides support for programming complex rules for accessing resources based on system metrics. Policies are declared in a special section of a Brainlet code named POLICIES. Each policy has a unique identifier and code implementing its behavior. In this way, the same rule can be referenced from more than one protocol, thus allowing reuse of policies.

On an m-failure, WS-Log searches for the resource instances that match the protocol of the predicate that m-failed. Then, if the protocol third argument references an existing policy, WS-Log evaluates the policy to decide the particular resource instance that will be accessed, and the particular access method that will be used. The programmer must specify these decisions by declaring two separate Prolog rules, both with the same identifier, and with the following format:

```
sourceFrom(Name,
           [ID_Res1, Site_1],
           [ID_Res2, Site_2], Result):- ...
accessWith(Name, [ID_Res, Site],
           MethodA, MethodB, Result):- ...
```

The first rule defines the logic to select the desired resource instance from a pair of candidates. Similarly, the second rule contains the behavior for choosing an access method given a pair of valid access methods (*MethodA* and *MethodB*). By *valid* we mean a method that WS-Log considers suitable for accessing a specific resource. For example, WS-Log does not consider a copy operation for accessing a large database. In addition, the global resource identifier is included as a parameter of each rule. Note that this feature is useful to specify constraints over the size, availability, owner, etc. of a resource.

We will extent the example of the file searcher Brainlet. Suppose for instance that agent has to use the following policy for accessing files: "access the file instance located at the nearest site. In addition, if the Brainlet size is less or equal than the file size, access the resource by using migration". The rules that define these policies are:

```
sourceFrom('file-policy', [_, Site_1], [_, Site_2], S):-
  proximity(Site_1, P_1),
  proximity(Site_2, P_2),
  min(P_1, P_2, Site_1, Site_2, S).
accessWith('file-policy', [ID_Res, Site], move, _, move):-
  agentSize(BrSize),
  resourceSize(ID_Res, FileSize),
  BrSize <= FileSize.
```

In this case, the rule *sourceFrom* estimates the proximity between the current agent location and each remote host, and then binds S with the address of the host that is closest to the local site. On the other hand, *accessWith* selects the move method for accessing a file provided the Brainlet size is less or equal than the file size. It is worth noting that *proximity*, *agentSize* and *resourceSize* are built-in predicates offered by the basic WS-Log policy programming support.

4.3 Runtime support

WS-Log is based on the RMF, a generic execution model able to automate decisions on when, how and what site to contact to satisfy agents resource needs. This mechanism is based on the idea that an entity external to the Brainlet helps it to find requested resources on an m-failure. In the RMF, those entities are stationary agents called PNS (Protocol Name Servers) agents.

Each host capable of executing and hosting Brainlets has one or more PNS agents. PNS agents are responsible for managing information about protocols offered at each site, and for returning the list of resource instances matching a given protocol under demand. A site offering resources registers with its local PNSs the protocols associated with these resources. Then, PNS agents announce the new protocols to other sites of the network by using a multicast-based communication mechanism.

Figure 2 depicts the WS-Log runtime support.When an m-failure occurs, the runtime support of WS-Log queries PNS agents for sites offering the needed resource, getting a list L_i of *hidden* properties (resource type, availability, size, etc.) of the instances matching the agent request. As pointed out before, hidden properties –unlike *public* ones– are not visible from protocols. However, they can be accessed by programmers through WS-Log built-in predicates in

This is a preprint of the article: "Extending MoviLog for Supporting Web Services. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley. Vol 33, Num. 1. pp 11-31. 2007. doi:10.1016/j.cl.2006.02.001"

The final publication is available at http://dx.doi.org/10.1016/j.cl.2006.02.001

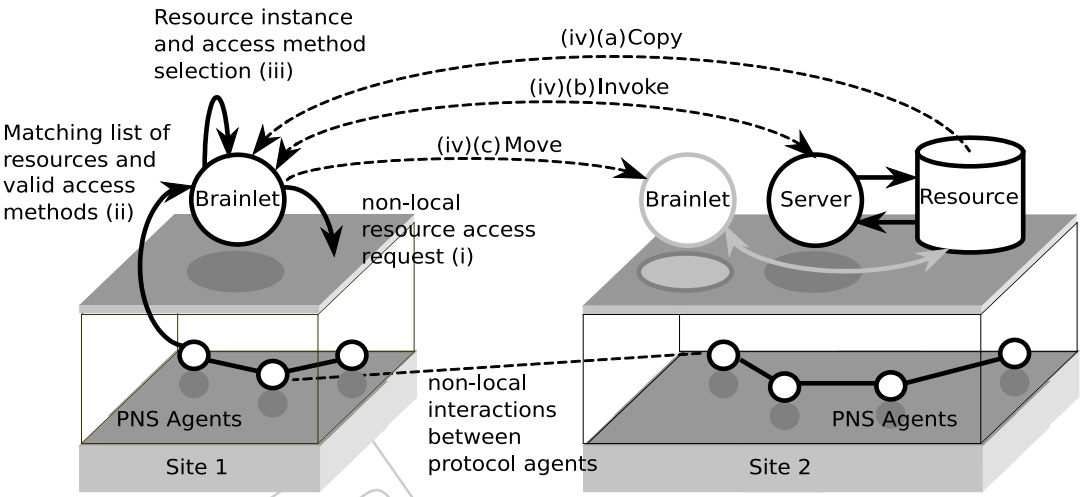

Figure 2. Overview of the RMF

order to specify custom resource access policies, as shown in subsection 4.2 in the policy for accessing files (file-policy).

Taking L_i as an input, WS-Log creates a list of pairs $L =< \alpha, \beta >$, where α represents the resource instance identifier, and β are the valid methods for accessing that instance. Based on the list L, the following tasks are performed:

(1) *Instance selection*: WS-Log selects from the input list the site from where the resource will be retrieved. In other words, an element from the list of instances is picked, leaving the remaining items as *backtracking* points to ensure completeness.
(2) *Access method selection*: WS-Log chooses the access strategy best suited for the current conditions (CPU load, network traffic, etc.) or particular requirements of the application (many or few interactions with the same resource). If defined, custom policies defined by the programmer are evaluated. Finally, the platform decides a method for accessing the resource instance.

The next section presents some details related to the design and implementation of WS-Log, focusing on the Web Service invocation support.

5 Design and implementation

The execution environment for Brainlets at a site is called a *Mobile Agent Resource* (MARlet). A MARlet is a Java servlet [32], which provides inference, communication and authorization services to Brainlets. Also, MARlets offer services for supporting the RMF by extending JavaLog [31], a Prolog-based language designed for programming stationary intelligent agents.

This is a preprint of the article: "Extending MoviLog for Supporting Web Services. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley. Vol 33, Num. 1. pp 11-31. 2007. doi:10.1016/j.cl.2006.02.001"

The final publication is available at http://dx.doi.org/10.1016/j.cl.2006.02.001

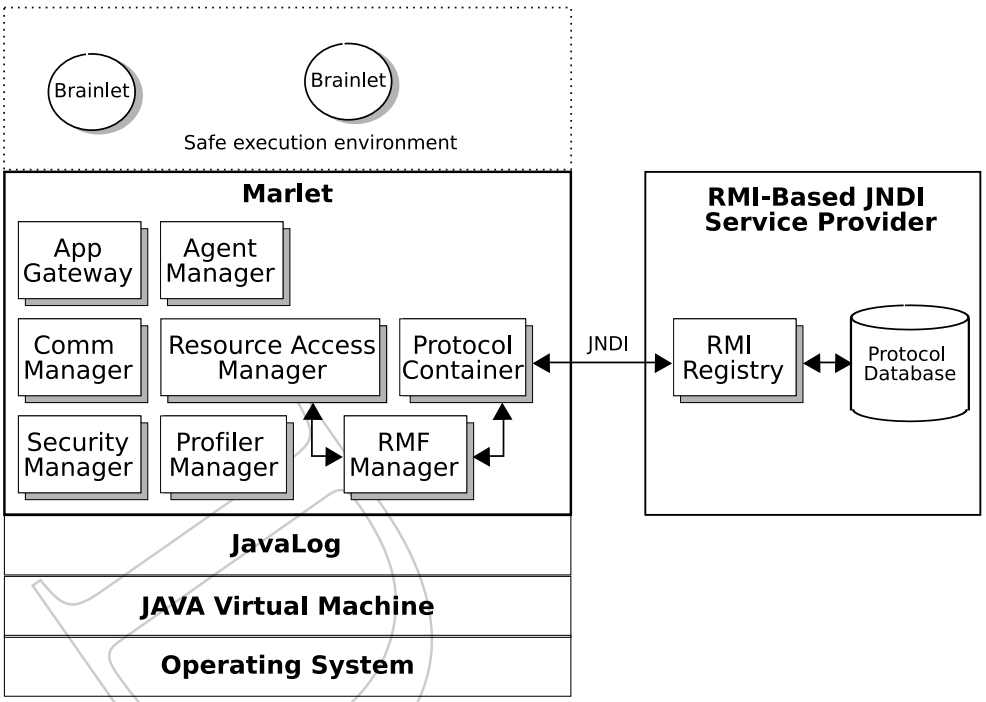

Figure 3. Architecture of the WS-Log platform

Figure 3 shows the architecture of a WS-Log site and its components. *Agent Manager* is in charge of marshaling agents into a network-transferable format, unmarshaling received agents and resuming their execution. *Security Manager* is responsible for validating agents code, and for granting access to resources according to preestablished security policies. The *Inter-Agent Communication Manager* offers message-based primitives for agent communication. The *Application Gateway* is a bridge between external applications and the agent-oriented services –inference, mental attitudes representation, communication, etc– offered by a site.

The components supporting the RMF are *RMF Manager*, *Profiler Manager*, *Resource Access Manager* and *Protocol Container*. The first one is responsible for managing and configuring PNS Agents, which in turn handle m-failures and protocol registrations and deregistrations. The *Profiler Manager* component estimates and maintains updated values of system execution conditions such as CPU load, free and occupied RAM, network transfer rate, state of outgoing and incoming communication links, among others. The *Resource Access Manager* checks and grants permissions when a specific method for accessing some resource is applied. Finally, the *Protocol Container* is responsible for storing the protocols and extra information about resources published by both the local and remotes sites.

The low-level communication with Web Services is supported by means of Axis, an open source implementation of SOAP provided by the Apache Software Foundation. Figure 4 shows the basic steps performed by WS-Log for ac-

This is a preprint of the article: "Extending MoviLog for Supporting Web Services. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley. Vol 33, Num. 1. pp 11-31. 2007. doi:10.1016/j.cl.2006.02.001"

The final publication is available at http://dx.doi.org/10.1016/j.cl.2006.02.001

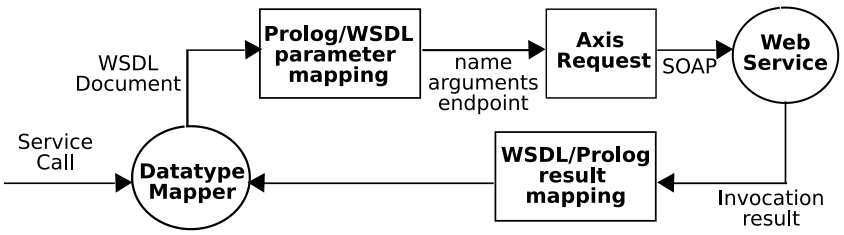

Figure 4. Web Service invocation in WS-Log

cessing Web Services. Upon an execution of a service operation operation, its associated WSDL document is parsed in order to extract arguments datatypes and map execution parameters from Prolog to their WSDL counterpart. Then, a SOAP request to the endpoint specified in the WSDL file is sent. Finally, the result(s) of the call are converted to Prolog structures and returned back to the RMF run-time.

The next section presents an application coded in WS-Log which shows how Brainlets interact with Web Services.

6 A sample application

In this section we show an application written in WS-Log. The application solves a constraint problem in the air travel domain [22]. Roughly, the goal of the application is to book an airplane ticket for traveling between two given cities according to certain domain business rules, such as for example flight and seats availability, and user preferences, such as constrains over the ticket cost or air companies. We assume the existence of Web Services for querying the availability of flights, seats, and costs, and for booking a flight.

In short, four Web Services are assumed to be available:

(1) *findFlight(Co, Origin, Destination, DepartureDate, ArrivalDate)*: Checks the existence of a flight in a company *Co* for the desired cities and dates. The service returns the ID of the first flight matching the criteria.
(2) *checkSpace(Co, flightID)*: Checks whether *flightID* has seats available.
(3) *checkCost(Co, flightID)*: Idem (2) for the cost of a flight.
(4) *bookFlight(Co, flightID)*: Books a flight and returns the ticket ID back to the client.

We consider a fixed set of companies and, in the initial state, the Brainlet implementing the application knows these companies. This knowledge is expressed in the form of Prolog facts *company(X)* stored in the Brainlet knowledge base. For simplicity, the universe of constraints a user may specify has been grouped in five categories, which led to five different variations of the

main problem:

- *Problem BPF (Book Preferred Flight)*: The Brainlet must book the flight offered by the user's preferred company whenever it is possible; otherwise, the agent should book a flight in any company.
- *Problem BMxF (Book Maximum Flight)*: This problem involves a different constraint: the user specifies a maximum price that he is willing to pay for a ticket.
- *Problem BPMxF (Book Preferred Maximum Flight)*: This problem considers not one but two user defined constraints, namely preferred company and maximum price, and can be seen as a mix between the two problems mentioned before.
- *Problem BBF (Book Best Flight)*: This variant proposes an optimization task where the user wants to book the cheapest flight available.
- *Problem BBPF (Book Best Preferred Flight)*: the Brainlet must book the cheapest flight available, but if two flights have the same price, the agent should favor the preferred company.

The following code implements a Brainlet for solving the BBPF problem. For space reasons, the implementations of the rest of the variants have been omitted. Nevertheless, BBPF is the hardest problem:

```
PROTOCOLS
  protocol(web-service, [operation-name(X),
    input(X)], none).
CLAUSES
  company('United␣Airlines').
  company('Air␣France').
  ...
  travel-info(['New␣York', 'Rome', 2005-12-31, 2005-01-02]).
  getCompaniesAndCost(I, CList):-
    getCompaniesAndCost(I, [], CList).
  getCompaniesAndCost(I, Temp, Result):-
    company(C),
    retract(company(C)),
    web-service([operation-name(findFlight),
    input([C|I])], FlightID),
    FlightID \= -1,
    web-service([operation-name(checkSpace),
    input([C|FlightID])], true),
    web-service([operation-name(checkCost),
    input([C|FlightID])], Cost),
    getCompaniesAndCost(I, [cost(C,FlightID,Cost)|Temp],
      Result).
  getCompaniesAndCost(_, Temp, Temp).
  minCost(cost(Co1,FID,Cost1), cost(_,_,Cost2),
```

```
        cost(Co1,FID,Cost1)):-
  Cost1 <= Cost2, !.
minCost(_, CostInfo2, CostInfo2).
findCheapest([CostInfo|Tail], Ch):-
  findCheapest(Tail, CostInfo, Ch).
findCheapest([CostInfo|Tail], TempCost, Ch):-
  minCost(CostInfo, TempCost, MinCost),
  findCheapest(Tail, MinCost, Ch).
findCheapest([], MinCost, MinCost).
selectMinCostCompany(CList, Cheapest, Pref, Cheapest):-
  not(member(cost(Pref,_,_), CList)).
selectMinCostCompany(_, PrefInfo, Pref, PrefInfo):-
  PrefInfo = cost(Pref, _, _).
selectMinCostCompany(CList, Cheapest, Pref, PrefInfo):-
  Cheapest = cost(_, _, Cost),
  member(cost(Pref,FID,Cost), CList),
  PrefInfo = cost(Pref, FID, Cost).
?-arrangeTravel(Pref, TicketID):-
  travelInfo(I),
  getCompaniesAndCost(I, CList),
  findCheapest(CList, Cheapest),
  selectMinCostCompany(CList, Cheapest, Pref, Result),
  Result = cost(Co, FID, _),
  web-service([operation-name(bookFlight),
    input([Co,FID])], TicketID).
```

The code contains one protocol for using a Web Service. As the reader can see, the only protocol specified does not define any special access policy, as indicated by the reserved word "none". Furthermore, the section CLAUSES defines the clauses which implement the algorithm for finding a ticket satisfying the preferences and constraints imposed by BBPF. The algorithm works as follows:

(1) First, it finds the company offering the cheapest flight.
(2) If the obtained company is equal to the one preferred by the user, the Brainlet books a ticket for that flight, and the execution finishes. After this, the agent returns to its origin if necessary.
(3) Similarly, if the company obtained from step (1) offers the same cost than the (later computed) preferred company, a ticket from this latter one is booked. Then, the Brainlet finishes its execution.
(4) Otherwise, the Brainlet books a seat for the resulting flight from step (1).

The potential points of activation of the RMF are the predicates whose functor is *web-service*. As these predicates first argument –with its functor– match a protocol definition, they do not represent a conventional Prolog call, but a Web Service access request. Therefore, the RMF will handle the evaluation of these

predicates whenever the service cannot be obtained at the local host.When a proper Web Service instance is located, the RMF contacts it by means of an Axis request that is built based on the WSDL document describing the service and the invocation parameters given by the agent. The following XML code shows a portion of the WSDL document used in the example:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<definitions name="ExampleServiceServer"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://server:port/example/FlightWS.wsdl"
    xmlns:tns="http://server:port/example/FlightWS.wsdl"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <message name="findFlightResponse">
    <part name="findFlightResult" type="xsd:int"/>
  </message>
  <message name="findFlightRequest">
    <part name="company" type="xsd:string"/>
    <part name="origin" type="xsd:string"/>
    <part name="destination" type="xsd:string"/>
    <part name="departureDate" type="xsd:date"/>
    <part name="arrivalDate" type="xsd:date"/>
  </message>
  ...
  <portType name="ServerPortType">
    <operation name="findFlight">
      <input message="tns:findFlightRequest"/>
      <output message="tns:findFlightResponse"/>
    </operation>
    ...
  </portType>
  <binding name="ServerSoapBinding" type="tns:ServerPortType">
    <soap:binding style="rpc"
       transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="findFlight">
      <soap:operation style="rpc"/>
        <input>
          <soap:body use="literal"
             namespace="urn:ExampleServiceServer"/>
        </input>
        <output>
          <soap:body use="literal"
             namespace="urn:ExampleServiceServer"/>
        </output>
    </operation>
    ...
  </binding>
```

```
  <service name="FlightBookingService">
    <port name="ServerPort" binding="tns:ServerSoapBinding">
      <soap:address
         location="http://server:port/axis/FlightBooking"/>
    </port>
  </service>
</definitions>
```

It is worth noting that the application described so far is based on the idea of utilizing Web Services. As explained, upon a client request, a mobile agent is asked to find a composition of *stationary* services which fulfills domain and user constraints. However, in some cases, the approach of having mobile agents providing mobile Web Services [7] is also interesting, since it permits mobile agents and Web Services to complement each other [23].

Note that a MWS-like [23] deployment of WS-Log agents is very simple. Each variant of the application shown could be thought as a separate Web Service which is in charge of the whole booking process. Furthermore, user preferences for each problem are mapped into input arguments for invoking these services. The logic for booking is implemented in WS-Log, since concrete implementations of services (i.e. their WSDL binding) must create, configure and launch a mobile agent in order to find and compose other services to solve the problem. Also, resource access policies for efficient interaction with services could be considered when creating those agents.

The final step in deploying our MWSs is publication. From a client agent or an external application perspective, a WSDL describing each service must be supplied, along with the proper bindings to mobile agents implementing application logic. From the WS-Log platform point of view, a protocol have to be created and announced so that others sites are aware of the service. This announcement is done by the platform right after a new Web Service is installed on a WS-Log site.

The following code represents a portion of the WSDL document describing our MWS. In this case, the binding denoted by *urn:BrFlightBookingServer* points to a JAVA class where each method corresponds to a specific WSDL operation (i.e. a booking variant). Within those methods, WS-Log agents are launched locally in order to solve the particular variant of the booking process based on the (possibly remote) simpler WSDL operations presented at the beginning of this section. In summary, our MWS would be composed of:

- a *stationary* part, given by the WSDL document describing the interface of the service's operations, and the JAVA binding, which acts as a bridge

[7] A mobile Web Service is a service with migration capabilities

between WSDL code and WS-Log mobile agents.

- a *mobile* part, consisting of various booking agents which potentially move to other sites in order to find and locally missing WSDL operations.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<definitions name="BrFlightBookingServer"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://server/example/BookingBrWS.wsdl"
    xmlns:tns="http://server/example/BrFlightBookingWS.wsdl"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <message name="BBPF-BookingResponse">
    <part name="BBPF-BookingResult" type="xsd:int"/>
  </message>
  <message name="BBPF-BookingRequest">
    <part name="preferredCompany" type="xsd:string"/>
    <part name="origin" type="xsd:string"/>
    <part name="destination" type="xsd:string"/>
    <part name="departureDate" type="xsd:date"/>
    <part name="arrivalDate" type="xsd:date"/>
  </message>
  ...
  <portType name="ServerPortType">
    <operation name="BBPF-Booking">
      <input message="tns:BBPF-BookingRequest"/>
      <output message="tns:BBPF-BookingResponse"/>
    </operation>
    ...
  </portType>
  <binding name="ServerSoapBinding"
    type="tns:ServerPortType">
    <soap:binding style="rpc"
       transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="BBPF-Booking">
      <soap:operation style="rpc"/>
        <input>
          <soap:body use="literal"
             namespace="urn:BrFlightBookingServer"/>
        </input>
        <output>
          <soap:body use="literal"
             namespace="urn:BrFlightBookingServer"/>
        </output>
    </operation>
    ...
  </binding>
  <service name="BrFlightBookingService">
```

```
    <port name="ServerPort"
      binding="tns:ServerSoapBinding">
      <soap:address
         location="http://server/axis/BrFlightBooking"/>
    </port>
  </service>
</definitions>
```

As one might expect, the client code for requesting a BBPF-like booking reduces significantly, since all logic for solving the BBPF problem is encapsulated on server-side Brainlets:

```
PROTOCOLS
  protocol(web-service, [operation-name(X), input(X)],
    none).
CLAUSES
  travel-info(['New␣York', 'Rome', 2005-12-31,
    2005-01-02]).
  ?-arrangeTravel(Pref, TicketID):-
    travelInfo(I),
    Input = input([Pref|I])
    web-service([operation-name(BBPF-Booking), Input],
      TicketID).
```

7 Experimental Results

In this section we report some results obtained from the experimentation with the application presented in the previous section. We compared the WS-Log solution to the air ticket problem with an implementation using IG-Jade-PKSLib [22], a toolkit for the development of multiagent systems for Web Service composition and provisioning.

The results of the IG-Jade-PKSLib implementation for the different variants of the flight booking problem are shown in table 2. These results were extracted from [33]. The experiments were performed on a XEON 3.0 Ghz with 4 Gb RAM, under Linux 2.4.22. The table shows the average execution time for five runs of each variant of the problem with a different number of companies varying from 2 to 10. All times are expressed in seconds, and $t_{max} = 300$.

In terms of performance, IG-JADE-PKSLib behaves reasonably well in BPF, BMxF and BPMxF, as solutions are generated in less than five seconds for five or less air companies. However, the implementation does not scale well for ten or more companies, as shown in figure 5. However, BBF and BBPF were

Table 2
Average execution time of IG-Jade-PKSLib

#Co	BPF	BMxF	BPMxF	BBF	BBPF
2	0.17	0.21	0.37	1.27	8.42
3	0.58	0.72	1.20	24.02	109.33
4	1.45	2.20	3.71	$> t_{max}$	$> t_{max}$
5	3.76	4.33	4.65	$> t_{max}$	$> t_{max}$
10	80.60	96.45	105.49	$> t_{max}$	$> t_{max}$

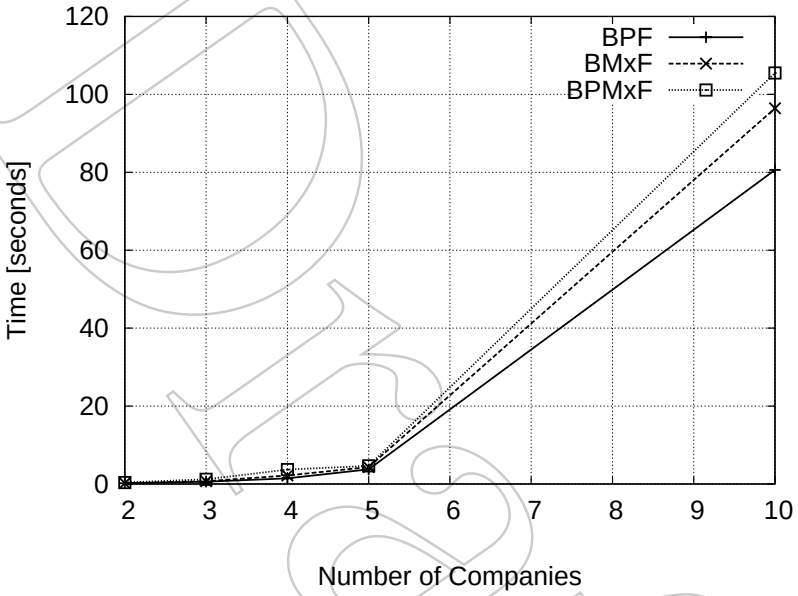

Figure 5. Performance of BPF, BMxF and BPMxF (IG-Jade-PKSLib)

too slow in the tests.

Table 3 summarizes the results for WS-Log. Figures 6 and 7 show the average execution time for each variant of the problem. In this case, the tests were conducted on a Pentium 4 2.2 Ghz with 1 Gb RAM, under Linux 2.6.8. The Web Services were deployed on the Apache Tomcat web server (version 4.1), running on a second machine under JDK 1.4.2 (build 05).

As the reader can see, WS-Log performed excellent, even when running on less powerful hardware. Unlike IG-JADE-PKSLib, all the solutions scaled well. As expected, the worst execution times were obtained from BBF and BBPF variants, because they are the most computationally demanding problems. Specifically, the Brainlet must find out the company that offers the least expensive ticket, which in turn requires checking prices in every company.

Table 3
Average execution time of WS-Log

#Co	BPF	BMxF	BPMxF	BBF	BBPF
2	0.38	0.44	0.47	0.49	0.53
3	0.43	0.49	0.48	0.58	0.62
4	0.43	0.50	0.52	0.65	0.63
5	0.45	0.51	0.55	0.67	0.81
10	0.48	0.61	0.72	0.84	1.02
20	0.69	1.05	0.99	1.40	1.39
50	1.11	1.78	2.01	2.38	2.70
100	1.45	3.22	3.60	3.75	4.01

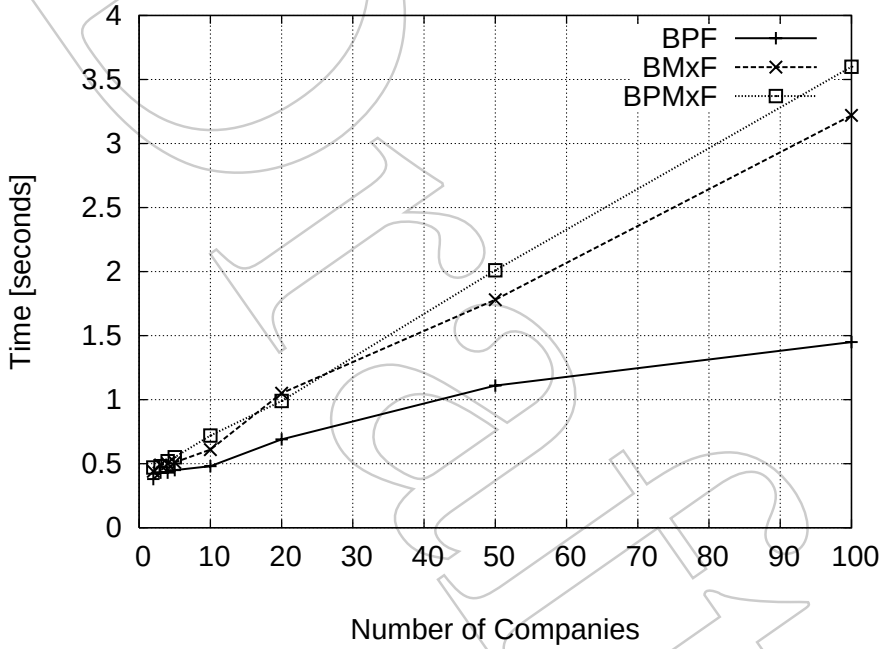

Figure 6. Performance of BPF, BMxF and BPMxF (WS-Log)

8 Conclusions

The WWW is increasingly evolving into an open medium that is not only a huge repository of Web pages and data, but also of services accessible through standard communication protocols. This represents a big step towards the realization of a new Web where applications use and share services and resources in a fully automated way.

In this computational environment, intelligent mobile agents will play a fun-

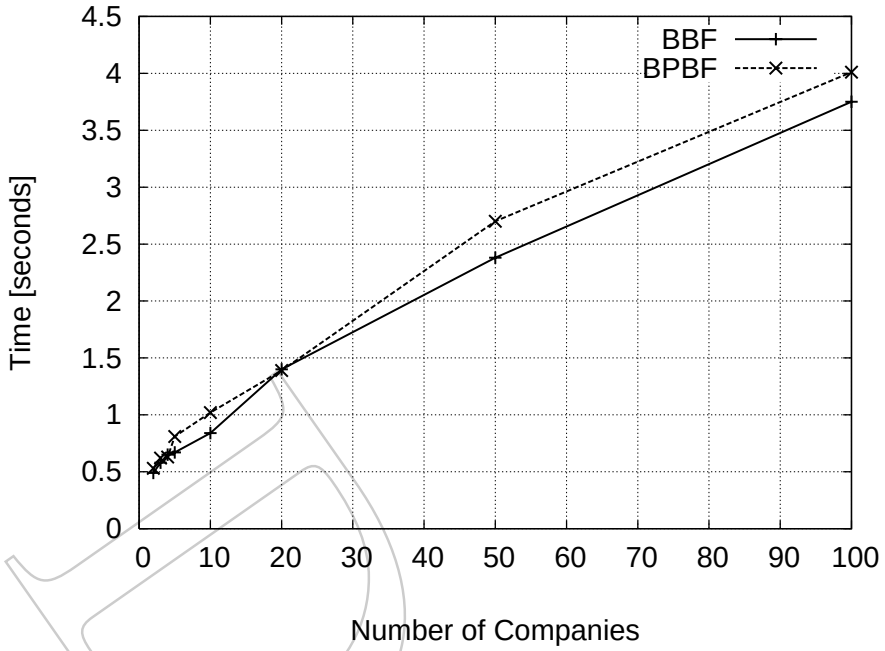

Figure 7. Performance of BBF and BBPF (WS-Log)

damental role because of their ability to infer, learn, act and move. In this sense, our research aims at providing languages and tools for building agents that interact with Web resources. We have developed WS-Log, a language for programming Prolog-based mobile agents integrated with Web Services. The main difference between WS-Log and other platforms for mobile agents is its support for reactive mobility by failure, which reduces development effort by automatizing mobility and resource access decisions. In addition, by using WS-Log it is easy to build agents that locate and invoke Web Services published across the WWW.

We have shown the practical usefulness of WS-Log through experiments. As reported, WS-Log performs very well with respect to related approaches.

The weak point of the approach for integrating WS-Log agents and Web Services is that it does not take into account the semantics of service operations. Currently, agents request operations by their name and the format of their input/output arguments. As a consequence, an agent is not able to obtain, for example, the complete list of operations for *car rental* if they have been published under a name different from the one supplied by the agent. Furthermore, significant programming effort must be done to process the output of operations whose functionality is the same, but their output format differs. We are exploring solutions to these problems in order to achieve a truly interoperability between WS-Log agents and Web Services. One approach to solve these limitations is the usage of machine understandable descriptions of the concepts involved in services. In this way, we are enriching WS-Log pro-

tocols and service discovery mechanisms with ontology support based on the technologies of the Semantic Web [27] and Semantic Web Services [3].

The current version of WS-Log has been tested with small applications. At present, we are implementing a workflow engine for document management based on mobile agents and Web Services. Also, we are building a paper submission and reviewing system with WS-Log agents and Semantic Web Services.

References

[1] T. Berners-Lee, Information management: A proposal, `http://www.w3.org/History/1989/proposal.html` (1989).

[2] W. Theilmann, K. Rothermel, Domain experts for information retrieval in the world wide web, in: CIA '98: Proceedings of the Second International Workshop on Cooperative Information Agents II, Learning, Mobility and Electronic Commerce for Information Discovery on the Internet, Springer-Verlag, London, UK, 1998, pp. 216–227.

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

[4] T. Berners-Lee, M. Fischetti, Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web, Harper San Francisco, 1999.

[5] F. Curbera, W. A. Nagy, S. Weerawarana, Web services: Why and how, in: Proceedings of OOPSLA 2001 Workshop on Object-Oriented Web Services, Tampa, Florida, USA, 2001.

[6] J. Martin, Web services: The next big thing, XML Journal 2 (5).

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

[8] B. Burg, Agents in the world of active web-services, in: Lecture Notes In Computer Science. Revised Papers from the Second Kyoto Workshop on Digital Cities II, Computational and Sociological Approaches, Springer-Verlag, 2002, pp. 343–356.

[9] J. Hendler, Agents and the semantic web, IEEE Intelligent Systems Journal 16 (2) (2001) 30–36.

[10] M. N. Huhns, Software agents: The future of web services, in: Agent Technology Workshops 2002, Vol. 2592 of Lecture Notes in Artificial Intelligence, 2003, pp. 1–18.

[11] A. R. Tripathi, N. M. Karnik, T. Ahmed, R. D. Singh, A. Prakash, V. Kakani, M. K. Vora, M. Pathak, Design of the ajanta system for mobile agent programming., Journal of Systems and Software 62 (2) (2002) 123–140.

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

[13] D. Kotz, R. S. Gray, Mobile agents and the future of the Internet, ACM Operating Systems Review 33 (3) (1999) 7–13.

[14] A. Zunino, M. Campo, C. Mateos, Simplifying mobile agent development through reactive mobility by failure, in: G. Bittencourt, G. Ramalho (Eds.), Advances in Artificial Intelligence, Vol. 2507 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 163–174.

[15] A. Zunino, C. Mateos, M. Campo, Reactive mobility by failure: When fail means move, Information Systems Frontiers. Special Issue on Mobile Computing and Communications 7 (2) (2005) 141–154, ISSN 1387-3326.

[16] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, Extensible markup language (xml) 1.0 (second edition) w3c recommendation, `http://www.w3.org/TR/REC-xml` (Oct. 2000).

[17] W3C Consortium, Simple object access protocol (soap) 1.1 specification, `http://www.w3.org/TR/SOAP/` (May 2000).

[18] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web services description language (wsdl) 1.1, W3C Note, World Wide Web Consortium . URL `http://www.w3.org/TR/wsdl`

[19] W3C Consortium, Uddi technical white paper, `http://www.uddi.org` (Sep. 2001).

[20] H. Kreger, Web services conceptual architecture (WSCA 1.0), Tech. rep., IBM Corporation (may 2001).

[21] S. A. McIlraith, T. C. Son, Adapting golog for programming the semantic web, in: Proceedings of the Fifth Symposium on Logical Formalizations of Commonsense Reasoning (CommonSense-01), New York, NY, 2001, pp. 195–202.

[22] E. Martínez, Y. Lespérance, IG-JADE-PKSlib: An Agent-Based Framework for Advanced Web Service Composition and Provisioning, in: Proceedings of the AAMAS-2004 Workshop on Web Services and Agent-Based Engineering, Morgan Kaufmann Publishers, New York, NY, 2004, pp. 2–10.

[23] F. Ishikawa, N. Yoshioka, Y. Tahara, S. Honiden, Towards synthesis of web services and mobile agents, in: Z. Maamar, C. Lawrence, D. Martin, B. Benatallah, K. Sycara, T. Finin (Eds.), Workshop on Web Services and Agent-Based Engineering (WSABE) (AAMAS'2004), New York, NY, USA, 2004.

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

[25] H. J. Levesque, R. Reiter, I. Lespérance, F. Lin, R. B. Scherl, GOLOG: A logic programming language for dynamic domains, Journal of Logic Programming 31 (1–3) (1997) 59–83.

[26] I. Horrocks, DAML+OIL: a description logic for the semantic web, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 25 (1) (2002) 4–9.

[27] T. Berners-Lee, J. Hendler, O. Lassila, The semantic web, Scientific American 284 (5).

[28] R. P. Petrick, F. Bacchus, A knowledge-based approach to planning with incomplete information and sensing, in: A. Press (Ed.), Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002), Menlo Park, CA, USA, 2002, pp. 212–221.

[29] R. E. Fikes, N. J. Nilsson, Strips: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (1971) 189–208.

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

[31] A. Amandi, M. Campo, A. Zunino, JavaLog: A framework-based integration of Java and Prolog for agent-oriented programming, Computer Languages, Systems and Structures 31 (1) (2005) 17–33, iSSN 1477-8424.

[32] J. Hunter, W. Crawford, Java Servlet Programming, O'Reilly & Associates, Inc., 1998.

[33] E. Martínez, Y. Lespérance, Web service composition as a planning task: Experiments using knowledge-based planning, in: Proceedings of the ICAPS-2004 Workshop on Planning and Scheduling for Web and Grid Services, Morgan Kaufmann Publishers, Whistler, British Columbia, Canada, 2004, pp. 62–69.

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.

Cristian Mateos is a Ph.D. candidate at the Universidad Nacional del Centro, working under the supervision of Marcelo Campo and Alejandro Zunino. He holds a System Engineer degree from UNICEN. He has implemented part of the runtime support for reactive mobility by failure in MoviLog. He is investigating the relationships between Web Services and mobile agents using reactive mobility by failure.

Alejandro Zunino received a Ph.D. degree in Computer Science from the Universidad Nacional del Centro (UNICEN), Tandil, Argentina, in 2003, his M.Sc. in Systems Engineering in 2000 and the Systems Engineer degree in 1998. He is a research fellow of the National Council for Scientific and Technical Research of Argentina (CONICET). He has published over 15 papers in journals and conferences. The main contributions of his 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 was 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 60 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*.