Expert Systems with Applications 38 (2011) 1723–1737

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

SWAM: A logic-based mobile agent programming language for the Semantic Web

Marco Crasso *, Cristian Mateos, Alejandro Zunino, Marcelo Campo

ISISTAN Research Institute, UNICEN University, Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina
Also Consejo Nacional de Investigaciones Cientficas y Tcnicas (CONICET), Argentina

ARTICLE INFO

Keywords:
Semantic Web
Ontologies
Web services
Mobile agents
Logic programming

ABSTRACT

Once a big repository of static data, the Web has been gradually evolved into a worldwide network of information and services known as the Semantic Web. This environment allows programs to autonomously interact with Web-accessible information and services. In this sense, mobile agent technology could help in efficiently exploiting this relatively new Web in a fully automated way, since Semantic Web resources are described in a computer-understandable way. In this paper, we present SWAM, a platform for building and deploying Prolog-based intelligent mobile agents on the Semantic Web. The article also reports examples and experimental results in order to illustrate as well as to assess the benefits of SWAM.

1. Introduction

The creation of the Web started early in the 90s. It quickly became popular among developers because it hid the diversity of software and hardware existing by then. This information space was designed to be fully distributed and without a central control. Basically, the Web maintains links or associations between the various documents that could be located and retrieved from any site of the Internet. The mechanism for browsing the Web is widely known: a user consults and interprets documents by reading HTML pages that are rendered by a special application, this is, the Web browser.

Years ago, the Web started to evolve into a worldwide network of annotated information and services (Shadbolt, Berners-Lee, & Hall, 2006). The objective of this new Web is to achieve automatic interaction between applications and Web resources. Particularly, Web Service technologies (Curbera et al., 2002; Vaughan-Nichols, 2002) provide the basis for standard ways of specifying well-defined, Web-accessible interfaces to access Web programs and resources. Basically, Web Services can be thought as a number of applications that interact by borrowing representational languages and transport protocols from established Internet technologies. In this sense, nowadays, the Web is not only concerned with information sharing but also with providing an evolved infrastructure for hosting programs that are autonomously exploited by user applications, including intelligent agents.

Web Services represent a suitable alternative to enable for systematic interactions of applications across the WWW. Web Services essentially rely on XML, a widely adopted structured language for information interchange that guarantees platform independence. Furthermore, on top of XML, several standard XML-based languages for invoking and describing services have been developed. WSDL (Curbera et al., 2002; W3C Consortium, 2007b) is an XML-based specification for describing Web Services as a set of callable operations over SOAP (Curbera et al., 2002; W3C Consortium, 2007a) messages, a high-level communication protocol also based on XML. From a WSDL specification, any user application can determine how to use the functionality an individual Web Service provides.

In the last past years, an increasingly number of Web Services have arisen, mainly in the context of e-commerce. For example, many popular Web sites such as Google,[1] Flickr[2] and eBay[3] offer Web Services for applications that expose the same information a user can access using a regular Web browser. Moreover, Amazon[4] delivers a set of Web Services that together interface a pay-per-use, reliable and scalable cluster computing platform for resource intensive applications. In addition, sites such as www.xmethods.com do not provide Web Services but offer a sort of "yellow page" by maintaining pointers to external services provided by other sites.

As the number of publicly available Web Services grows, several registries of services spring. A service registry represents a crossroad in the path of providers and consumers. Providers can

* Corresponding author at: ISISTAN Research Institute, UNICEN University, Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel./fax: +54 2293 440363.
E-mail address: mcrasso@conicet.gov.ar (M. Crasso).

[1] Google Code http://code.google.com.
[2] Flickr Services http://www.flickr.com/services/api.
[3] eBay http://developer.ebay.com/DevProgram/index.asp.
[4] Amazon Web Services http://aws.amazon.com.

0957-4174/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2010.07.098

use the registry to publish their services, while consumers can use it to find services that match their needs. After discovery, a consumer must bind (i.e. to connect) his application to the (usually remote) Web Service in order to interact and interchange data with it. UDDI (Curbera et al., 2002; OASIS Consortium, 2004) defines a standard mechanism for searching and publishing WSDL-described Web Services. By means of UDDI, a Web Service provider registers information about the services he offers, thus making them available to potential clients. The information managed by UDDI ranges from WSDL documents describing services interfaces to data for contacting service providers (e.g. location, email addresses, etc.). Basically, UDDI standardizes and extends the idea behind sites such as www.xmethods.com to offer service browsing capabilities to users.

Currently, the above "publish-find-bind" process is fully handled by human developers. Unfortunately, the "find" and "bind" operations still present many limitations that hinder the adoption of Web Services. On one hand, common materializations in the software industry for service registries do not supply developers with a full-featured discovery support. For example, UDDI supports only keyword-based search and category browsing of Web Services. This limitation is also present in contemporary keyword-based, service search engines such as `seekda.com`. Clearly, finding proper services – i.e. those fulfilling the *functional* expectations of the client – through UDDI is a time-consuming task when the number of services is large, which is the case of massively distributed environments such as the Web. Furthermore, *binding* an application to a Web Service requires developers to interpret its associated WSDL description and to provide the necessary boilerplate code to programmatically contact the service and execute its operations. When following this approach to Web Service invocation, developers are responsible for obtaining the service endpoint and datatype definitions, and employing low-level communication libraries for consuming the service. Alternatively, developers can employ frameworks for invoking Web Services, such as the DAIOS (Leitner, Rosenberg, & Dustdar, 2009) or the CXF (Apache Software Foundation, 2009), which provide programming abstractions to deal with Web Service access and invocation. In the end, developers must not only literally decipher each service's intended purpose and invocation details but also prepare their applications to consume selected services.

In order to facilitate the connection of Web Service consumers and providers, there is an increasing need for automating the way programs interact with services. Specifically, the focus should be on the "find" and "bind" operations. First, to find proper services automatically, a novel direction proposes to enhance Web Service descriptions using a non-ambiguous and computer-understandable format. By assuming that services are precisely described, it is expected that any application can autonomously understand the concepts involved within the set of tasks a Web Service performs or even the contents of a Web information source. Precisely, the Semantic Web effort proposes to annotate service descriptions with non-ambiguous concept definitions from shared ontologies (Martin et al., 2007; Paolucci, Kawamura, Payne, & Sycara, 2002). This allows applications to autonomously understand, from such semantically enhanced Web Service descriptions, the functional capabilities of any Web Service and the involved interaction mechanisms it prescribes (e.g. protocols). The idea behind the Semantic Web is fairly simple: every Web resource (services and information sources such as pages, files, databases and so on) is annotated with precise descriptions of its semantics, known as metadata. Applications then use these metadata to *understand* the properties and the capabilities of those annotated Web resources. These annotations are often expressed in standard semantic languages such as RDF (W3C Consortium, 2004) and OWL (Antoniou & van Harmelen, 2003).

The intelligent agent paradigm has been historically conceived to have a fundamental role in materializing the vision of user applications that autonomously understand such metadata (Hendler, 2001). Particularly, software agents (Hendler, 2001) – this is, autonomous software programs that perform tasks on behalf of users – can exploit the semantics of Web Services to supply consumers' applications with the knowledge necessary to bind them to external Web Services automatically. Moreover, due to the massively distributed nature of the Semantic Web, mobile agents (or agents able to migrate within a network to interact with locally accessible resources (Fortino, Garro, & Russo, 2008)) have properties that make them even more suitable for exploiting the potential of open information environments. In fact, mobile agents have been proposed for exploiting Cloud infrastructures (Douglis, 2008), a recent model for highly scalable distributed computing. Some well-known advantages of mobile agents with respect to ordinary agents are support for disconnected operations, heterogeneous systems integration, robustness and scalability (Lange & Oshima, 1999).

Despite the advantages mobile agents offer, many challenges remain in order to glue them with Web Services technology. Still, most of these challenges are a consequence of the nature of the WWW, since from its beginnings Web content has been mainly designed for human use and interpretation (McIlraith, Son, & Zeng, 2001). In other words, there is a need for a proper support for semantic service discovery for mobile agents. In addition, there is still a lack of proper programming mechanisms so that mobile agents can autonomously take advantage of the capacities of Web Services and resources. These facts, together with the inherent complexity of mobile code programming compared to traditional non-mobile systems, have hindered the massive adoption of mobile agent technology and limited its usage to small applications and academic prototypes (Douglis, 2008).

In this sense, we believe there is a need for a mobile agent development infrastructure that addresses these problems and, at the same time, preserve the key benefits of mobile agent technology for building massively distributed applications (Douglis, 2008). To cope with this, we propose SWAM, a platform for building and deploying Prolog-based mobile agents on the Semantic Web. SWAM defines a mobile agent execution model that allows programmers to easily create and deploy mobile applications without worrying about Web Services location or access details. Furthermore, in order to consider the semantics of services, SWAM provides an infrastructure for semantic matching and discovery of Semantic Web Services (Mateos, Crasso, Zunino, & Campo, 2006). This infrastructure aims at enabling for a truly automatic interoperability between SWAM agents and Semantic Web Services along with little development effort. As we will explain later, Prolog is central to our approach, since it has been recognized as an excellent choice for developing intelligent agents as well as exploiting semantic information.

This article is organized as follows. The next section describes SWAM, focusing on its syntax and its mobile agent execution model. Section 3 describes the semantic discovery subsystem of SWAM. Section 4 presents an example application to illustrate the interaction model between mobile agents and Semantic Web Services promoted by SWAM. Section 5 reports an evaluation of SWAM. Section 6 discusses relevant related works. Finally, Section 7 presents concluding remarks and future works.

2. SWAM

SWAM (**S**emantic **W**eb-**A**ware **M**oviLog) is a language for programming Prolog-based mobile agents and deploying them in the

Semantic Web. SWAM is built upon an extension and a generalization of MoviLog (Mateos, Zunino, & Campo, 2007; Zunino, Mateos, & Campo, 2005). MoviLog is a platform for building intelligent mobile agents on the WWW following a strong mobility model (Milanés, Rodriguez, & Schulze, 2008), where agents' execution state is transferred transparently on migration. Besides providing basic strong mobility primitives, an interesting feature of SWAM is the notion of Reactive Mobility by Failure (RMF) (Zunino et al., 2005), notion not exploited by any other tool for mobile agents. Conceptually, a *failure* is defined as the impossibility of an executing mobile agent to find some required resource at the current site (Zunino et al., 2005).

SWAM execution units are mobile agents called *Brainlets*. Each Brainlet carries Prolog code that is organized in two sections: *protocols* and *clauses*. The former section declares rules that are used by RMF for managing mobility. Specifically, protocols are the interfaces or descriptions of those resources which may trigger RMF-like mobility. On the other hand, the clauses section defines agent behavior and private data. Syntactically, the code of a Brainlet has the following form:

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

RMF states that when a predicate declared in the PROTOCOLS section of an agent fails, SWAM moves the Brainlet and its execution state to a site that contains definitions for the predicate and then resumes the Brainlet's execution. 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 has been declared are treated by RMF so that their failure may cause migration. To distinguish between Prolog failures with the traditional semantics and failures handled by RMF, we will refer to the latter as *m*-failures.

Let us take a closer look at the SWAM language. For instance, the following code:

```
PROTOCOLS
  protocol (aFunctor, [arity (2)]).
CLAUSES
  anotherFunctor (Y):-...
  aQuery:-aFunctor (X,Y), anotherFunctor (Y).
  ?-aQuery.
```

implements a Brainlet whose behavior is governed by the rules included in the CLAUSES section. Section PROTOCOLS states that every clause whose functor is "**aFunctor**" and arity is two will be treated by RMF. In this way, when the evaluation of aFunctor (X,Y) fails, the agent will be transferred to a site that contains definitions for that clause. Then, in case of a successful evaluation of aFunctor (X,Y) at the remote site, the agent will attempt to solve anotherFunctor (Y) according to the standard Prolog evaluation semantics; otherwise the evaluation of ?-aQuery will fail, because of the failure of aFunctor (X,Y).

Next is another example, presenting a Brainlet whose goal is to collect temperature values from different distributed sites and then to calculate the average of these values. Each measurement point is represented by a site with a sensing process that periodically stores the last measurement Value in a local database as a temperature (Value,Unit) predicate. The SWAM code implementing the Brainlet is:

```
PROTOCOLS
  protocol (temperature,[arity (2)]).
CLAUSES
  % Computes average and performs unit conversion
  average (List, Avg):-...
  % Collects temperature values
  getTemp (Curr, List):-
    temperature (Value, Unit),
    currentSite (S),
    not (member (measure (Value,_,S), Curr)),
    getTemp ([measure (Value,Unit,S)|Curr], List).
  % All sites have been visited
  getTemp (Curr, Rev):-reverse (Curr, Rev).
  average (Avg):-
    getTemp ([], List),
    average (NewList, Avg).
  % Brainlet's main goal

?-average (Avg).
```

The idea of the program is to force the Brainlet to visit all available sites, locally getting on each site the last measured temperature. The potential activation point of RMF is the temperature (Value,Unit) predicate. PROTOCOLS declares that the evaluation of temperature (Value,Unit) must be handled by RMF. As a consequence, if the evaluation of this predicate fails at a site S, RMF will move the Brainlet to a site containing definitions for temperature/2 (i.e. predicates with functor "temperature" with two arguments). The evaluation of getTemp will end successfully once all the sites offering temperature/2 have been visited.

For the sake of explaining the execution of the program, we will consider a network comprised of three SWAM-enabled sites. The idea is to trigger mobility upon *m*-failures of predicates temperature/2 and therefore forcing the Brainlet to visit the three sites S_1, S_2 and S_3. 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 evaluates temperature for the second time. In this case, the evaluation of temperature fails because the value stored at S_1 has been already collected. Considering that temperature has been declared as a protocol, an *m*-failure occurs. Then, RMF searches for sites providing temperature/2 so as 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 RMF selects S_2. Then, after the migration to S_2, getTemp collects the local temperature until no more choices are available. At this point, another *m*-failure occurs and RMF selects S_3. After evaluating once at S_3, temperature *m*-fails again. As there are not more options left for migrating the agent, evaluation of getTemp ends. Then, the average of the values [temperature ($Value_1, Unit_1$), temperature ($Value_2, Unit_2$), temperature ($Value_3, Unit_3$)] is computed, thus causing the evaluation of ?-average (Avg) to finish. Finally, the agent is automatically returned[5] to its origin (S_1).

In the example, the agent visits all sites containing temperature values. This behavior is not forced by SWAM, but by getTemp, because it evaluates all available temperature predicates so as to make not(member(...)) true. In other words, when an *m*-failure occurs, RMF moves the mobile agent to one particular site, leaving remaining alternatives as backtracking points.

It is worth noting that SWAM does not restrict the programmer to make use of RMF for handling mobility. Instead, by declaring protocols the programmer is able to select which predicates of a Brainlet's code can trigger mobility. At an extreme, a Brainlet

[5] An agent is sent back to its origin when it finishes its entire execution, regardless the execution is successful or not.

may not declare any protocol. This does not imply that mobility is not available, but it is in charge of the programmer as in most languages for mobile agent programming. Indeed, protocol declarations allow the programmer to use RMF and traditional proactive mobility at the same time, depending on his requirements.

The next subsection explains the underlying execution model of SWAM in detail. Then, Section 2.3 describes its policy programming support.

2.1. Generalized reactive mobility by failure

RMF (Zunino et al., 2005) was mainly designed to automate mobility decisions such as when and where to move a Brainlet. However, blindly moving an agent every time some required resource is not locally available can lead to situations where performance is bad (Zunino, Mateos, & Campo, 2005; Mateos et al., 2007). An illustrative example of this fact arises when 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. Another example takes place when the requested resource is a Web Service, because a transfer is not feasible, and hence the proper way to use it is by invoking the service, thus only inputs and outputs are transferred. Finally, the interaction of an agent with a large database can be better done by moving the agent to the provider site, and then locally interacting with the data. In this case, database access by copy is unacceptable because it might use too much network bandwidth.

SWAM improves RMF by defining a new execution model named GRMF (Generalized Reactive Mobility by Failure) which includes extra methods for accessing resources besides agent mobility. GRMF supports proper methods for interacting with Web resources, such as remote invocation, for the case of Web Services, and mirroring of resources, for the case of data and code. From the point of view of the mobility model, GRMF generalizes RMF, since remote invocation implies control flow migration, and replication can be considered as a form of resource migration. From the resource access point of view, GRMF extends RMF. GRMF provides decision mechanisms for selecting an access method based on both resource types and environmental conditions, such as the total free memory available at a given site, the network transfer rate and so on.

When an *m*-failure occurs, there may be many sites offering the needed resource. SWAM is able to decide an ordering for accessing sites if a Brainlet needs visiting more than one of them. In addition, since depending on the nature of a resource several access methods may be suitable, SWAM can apply different tactics to select the most convenient one. Both, ordering sites and choosing an access method are decided by SWAM through *policies*. Policies are decision mechanisms based on platform-level metrics such as network traffic, nearness between sites, agent size, CPU load, among others. For example, one may specify that any access to a certain large database should be done by moving the mobile agent where the database is located, rather than performing a potentially time and bandwidth-consuming copy operation of the required data from the remote site. Besides, SWAM lets the programmer to define custom policies for adapting GRMF to fit specific application requirements.

2.2. Protocols

As explained before, protocols are descriptors or logical pointers a Brainlet uses to reference the set of resources it could need along its lifetime. Protocols must be declared in the PROTOCOLS section of a Brainlet code with the following Prolog structure:

$$protocol(resourceKind, [prop_1, prop_2, \ldots, prop_n], accessPolicy)$$

where:

- *resourceKind* is a literal (Prolog atom) representing the category which the resources referenced by the protocol belong to. A category stands for any kind of resources made accessible to Brainlets, such as files, databases, code libraries, Prolog clauses and Web Services. For example, protocol(file,[name(index.html)],_) refers to all resources of type "file" whose name is "index.html".
- The second argument of the protocol corresponds to the list of properties the desired instances must match, which allows a Brainlet to reference different subsets from the set of resources belonging to the *resourceKind* class. Each property is a Prolog fact *A*(*B*), where *A* is the property name, and *B* contains the property value (e.g. the pair "name" and "index.html" in the previous example).
- Finally, *accessPolicy* contains the identifier of the policy used by the agent to choose a 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 a POLICIES section, as will be explained later. A "none" value indicates that all access decisions over the referenced set of resources are fully delegated to GRMF.

As one might expect, every resource kind defines its own set of properties for describing resource instances. For example, a protocol for a Web Service resource should include the operation name,[6] input arguments and outputs. The following SWAM code declares a protocol describing a Web Service for searching Web pages based on one or more keywords:

```
protocol (webService,[operation (keywordSearch), in
([keywords (K)]),out (_)],none)
```

Roughly, the previous declaration enables GRMF to act whenever a generic search service cannot be found at the local site. When this occurs, GRMF searches for published Web Services including an operation whose name matches "keywordSearch", such as the ones provided by Google or Amazon. Also, the search is constrained to those operations with one argument (keywords) but does not restrict the operation output (i.e. the Prolog fact representing the search result). Moreover, since no access policy has been specified for the protocol, the runtime of GRMF is in charge of selecting an appropriate method for contacting services. Note that a different protocol for a specific search pattern – such as a hardwired list of keywords – could be declared, just by replacing the variable K with the desired values. This is useful for applying different user-defined policies for accessing different subsets of service instances.

As the reader can see, some attributes of the Web Service such as the server address or the transfer protocol are not specified. The information needed for contacting a search service instance is encapsulated in its associated WSDL file and is extracted and used by SWAM at runtime when a specific instance is selected. We call this kind of attributes the *hidden* properties of a resource, this is, descriptive information accessible only to the SWAM platform and therefore not taking part in the protocol matching process. Hidden properties and *public* properties (WSDL document location; operation, in and out in our example, respectively) must be supplied by providers when they publish a service or a resource in SWAM.

One of the main benefits of consuming Web Services using SWAM is that it deals with seamlessly incorporating external

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

services into users' applications. Upon a Web Service that fits a protocol declaration is located, SWAM interprets its associated WSDL document and invokes its functionality. Besides freeing developers of providing the necessary boilerplate code to consume a service, SWAM shields service consumers from all non-functional aspects of service binding.

One of the main limitations of the aforementioned support for consuming Web Services, on the other hand, is that having syntactic-based protocol specifications requires that consumers and providers use the same signature to request and offer services, respectively. Obviously, this is a very brittle approach in decentralized environments like the WWW. In the end, only those services that exactly match the protocol description can be used. In order to overcome this restriction, we exploit semantics-based descriptions of services and requests, or protocols in the context of GRMF. SWAM introduces another kind of resources named *semanticWebService*, which stands for Web Services which have been semantically described using shared ontologies. To indicate the need of a *semanticWebService* resource, a developer should write the properties of the expected service using concepts from the same shared ontologies. For example, let us suppose that a developer uses the concepts "s:KeywordBasedSearch" and "s:KeywordList" to semantically describe the operation and inputs of a desired service, then the code of the protocol would be:

```
protocol (semanticWebService, [operation ('s:Key-
wordBasedSearch') in ('s:KeywordList'), out (_)],
none)
```

By assuming that both publishers' services and (agent) discoverers' protocols are precisely described, it is expected that connecting them would be simplified. Clearly, one limitation of this semantics-based approach to *matching* protocols and queries is that it requires to describe them using the same concepts. Section 3 will present a semantic similarity scheme that addresses this limitation.

Another example is presented next, which consists of a mobile agent for distributed text file search. In particular, the Brainlet has to find the files whose name is "Book.html" containing the string "Semantic Web". For simplicity, wild cards are not supported. The code that implements the agent is (let us ignore for the time being the contents of the POLICIES section):

```
PROTOCOLS
  protocol (file, [name(X)], none).
POLICIES
  % empty section
CLAUSES
  searchForFiles (FileName, Text, Files):-
    assert(filesFound ([])),
    findFiles (FileName, Text),
    retract(filesFound (Files)).
  findFiles (FileName, Text):-
    file([name(FileName)], FileProxy),
  analizeFile (Text, FileProxy), !, fail.
  findFiles (_,_).
  analizeFile (Text, FileProxy):-
  searchText (Text, FileProxy),
  getURI (FileProxy, FileURI),
    saveResult (FileURI).
  % asserts a new result
  saveResult (FileURI):-
    retract(filesFound (Temp)),
    assert(filesFound ([FileURI|Temp])).
  % Checks whether FileProxy contains Text
  searchText (Text, FileProxy):-...
  ?-searchForFiles (''Book.html",''Semantic Web",
  Result).
```

In this case, the agent defines one protocol that declares the need for accessing, at some point of its execution, one or more instances of a "file" resource. The protocol indicates that those instances must have their "name" attribute as a public property. Every time the agent evaluates a rule requiring a file, GRMF will handle the request.

The example shown includes a new section named POLICIES. This is the place where programmers can define custom heuristics for resource retrieval (in this case, files). For example, it could be convenient to select the access method according to the file size. If it exceeds a certain size, the agent could 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 explain how to state these decisions.

The Brainlet begins its execution by evaluating ?-searchForFiles. When a predicate fails accessing a file, SWAM asks the current site for the list of remote resource instances matching the protocol requested. To be more specific, SWAM searches for those instances which have been tagged with the "file" resource category with a public "name" property. As the agent does not declare any policy for accessing files, SWAM chooses any instance from the list and a proper access method. When the file predicate is reevaluated, SWAM uses another item of this list. Once all items have been consumed, the predicate cannot be further reevaluated, and the file searching process ends. After this, the agent is moved to the site from where it was initially launched.

The actual access to every file instance is requested through the predicate file (name (FileName), FileProxy), which filters via standard Prolog unification those files whose name is not the same as FileName. Moreover, the FileProxy variable is instantiated with a platform-supplied object that hides the real location of the file and provides a handler to read its contents.

2.3. Customizing GRMF

SWAM allows programmers to customize the way mobile agents interact with resources to fit specific applications particularities. SWAM provides support for programming complex rules for accessing resources based on platform-level metrics. Policies are declared in a special section of a Brainlet's 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.

Upon an *m*-failure, SWAM searches for the resource instances that match the protocol of the predicate that *m*-failed. Then, if the protocol references an existing policy, SWAM evaluates this policy to decide the particular resource instance that will be accessed, and the particular access method that will be used. The programmer specifies these decisions by declaring two separate Prolog rules, both with the same identifier and with the following structure:

```
sourceFrom (PolicyName, resource (Id1, Host1),
              resource (Id2, Host2), Result):-...

accessWith (PolicyName, resource (Id, Host),
              MethodA, MethodB, Result):-...
```

The first rule defines the logic to select the desired resource instance among any pair of candidates. Similarly, the second rule contains the behavior for choosing an access method given any pair of valid access methods (MethodA and MethodB). By valid we mean a method that SWAM considers suitable for accessing a specific kind of resource. For example, SWAM does not consider a copy operation for accessing a large database. In addition, the global resource identifiers (i.e. Id1 and Id2) are included as arguments of each rule. Note that this feature is useful to obtain information about a resource and thus to specify constraints over its size, availability or cost.

We will extend the example of the file searcher Brainlet previously discussed. Suppose for instance that the agent has to use the following policy for accessing files: "access the file instance located at the host with the best bandwidth. In addition, if the Brainlet size is less or equal than the file size, access the file via migration". The rules that are necessary for coding this policy are:

```
sourceFrom(#1, resource(_, Host1), resource(_, Host2),
  H):-
  transferRate (Host1, T1),
  transferRate (Host2, T2),
  minimum (T1, T2, Host1, Host2, H).
accessWith(#1, resource (Id, Host1), move,_, move):-
  agentSize (AgentSize),
  resourceSize (Id, FileSize),
  AgentSize<=FileSize.
```

The rule sourceFrom estimates the transfer rate between the current agent location and each remote host and then binds H with the address of the host to which the local site experiments the best network transfer rate. On the other hand, accessWith selects the move method for accessing a file provided the Brainlet's size is less or equal than the file size. It is worth noting that transferRate, agentSize and resourceSize are built-in predicates offered by the policy support of SWAM, which also provides various metrics related to environmental conditions such as CPU, memory and storage availability.

2.4. Runtime support

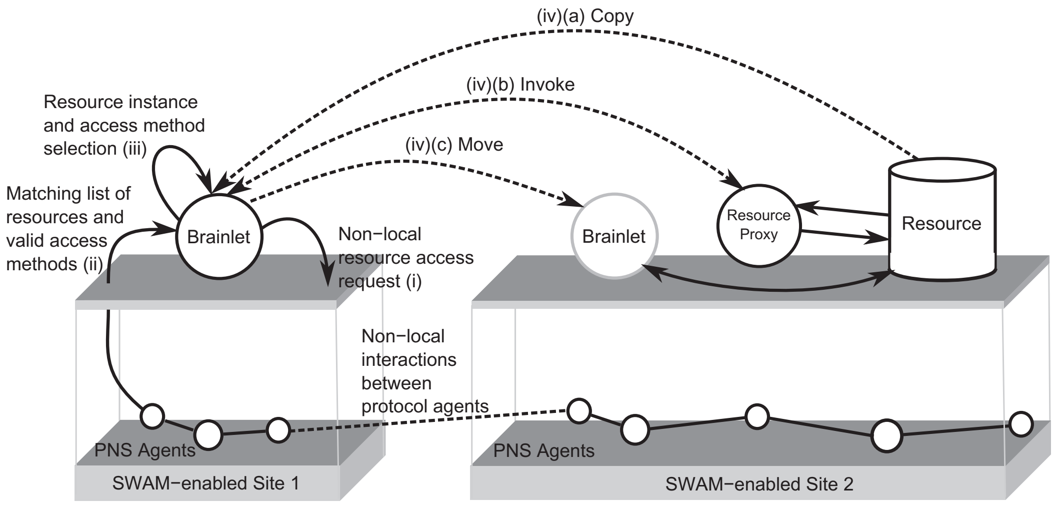
SWAM is based on GRMF, a generic execution model able to automate decisions on when, how and what site to contact to satisfy agents resource needs. GRMF is based on the idea that an entity external to the Brainlet helps this latter to handle *m*-failures. Those entities are stationary agents called PNS (Protocol Name Servers) agents.

Each site capable of hosting Brainlets (i.e. a SWAM-enabled site) 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. For example, when publishing a Web Service, the protocol-related information associated with the service – i.e. operations, valid inputs and delivered outputs – must be supplied by the user to the hosting SWAM site. As a consequence, the local PNS agents announce the newly added protocol to other sites of the network. Announcement is performed by means of GMAC (Gotthelf, Zunino, Mateos, & Campo, 2008), a communication protocol of our own specially designed to provide efficient multicast messaging to mobile platforms in open environments.

Fig. 1 depicts the SWAM runtime support. When an *m*-failure occurs, SWAM queries local PNS agents for sites offering the needed resource (step (i) in the figure), getting a list L_i of *hidden* properties (resource size, provider host, etc.) of the instances matching the agent request (step (ii)). As pointed out before, hidden properties – unlike *public* ones – are not visible from protocols. In other words, protocol declarations are not allowed to include properties such as "size", "host", "availability" and so on. However, hidden properties can be accessed by programmers through SWAM built-in predicates in order to specify custom resource access policies, as shown in Section 2.3 through the policy for accessing files. Note that, in that case, the file size is queried by means of a "resourceSize" built-in predicate.

Taking L_i as an input, SWAM creates a list of pairs $L = \langle\alpha, \beta\rangle$, 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 (step (iii)):

1. *Instance selection*: SWAM selects from the input list the site from where the resource will be retrieved. In other words, an item from the list of instances is picked, leaving the remaining items as backtracking points to ensure completeness. If defined, the policies coded by the programmer are evaluated.
2. *Access method selection*: SWAM chooses the access strategy taking into account the current platform-level execution conditions such as CPU load and network traffic. If present, policies declaring custom access strategies are also evaluated.



(iv)(a) Copy
(iv)(b) Invoke
(iv)(c) Move
Resource instance and access method selection (iii)
Matching list of resources and valid access methods (ii)
Brainlet
Non−local resource access request (i)
Brainlet
Resource Proxy
Resource
Non−local interactions between protocol agents
PNS Agents
SWAM−enabled Site 1
PNS Agents
SWAM−enabled Site 2

Fig. 1. Overview of GRMF.

Finally, the platform uses the selected method for accessing the resource instance as shown in steps (iv)(a), (iv)(b) and (iv)(c) of Fig. 1. These steps are sketched using dashed lines to denote that only one of them is performed.

Up to this point, we have explored the programming and execution models of SWAM for implementing mobile agents, focusing on the application-level and the middleware-level mechanisms by which agents indicate and contact external resources that are required to perform their tasks. Particularly, resource needs are indicated by means of protocols, this is, structured descriptions of the properties of external resources such as files, libraries and services. In order to effectively access Web Service resources we have implemented a support for semantic discovery, which addresses the tasks of managing, querying and reasoning about protocols and Web Service metadata. The next sections focus on describing this support.

3. Semantic matching in SWAM

Semantic matching allows agents to take advantage of ontologies by using reasoners. An ontology explicitly represents the meaning of terms in vocabularies and the relationships between these terms (Berners-Lee, Hendler, & Lassila, 2001). Moreover, a reasoner is a software component that infers knowledge that is implicitly conveyed in ontologies. For example, let us suppose we have an ontology representing the relationships "John is Mary's father" and "Mary is Ben's mother". Then, if we supply a reasoner with the rules "father $(x,z) \wedge$ parent $(z,y) \rightarrow$ grandFather (x,y)" and "mother $(x,y) \cup$ father $(x,y) \rightarrow$ parent $(x,-y)$", the reasoner would infer that "John is Ben's grandfather" from the ontology.

Indeed, one of the most powerful uses of ontology-based reasoning, and a key enabler for agents on the WWW, is in the area of Web Services. Old approaches to managing machine-to-machine interactions over the Web have been mainly focused on providing standard to make sure an agent knows the interface and invocation details of a service before the interaction actually takes place. However, this is very inflexible, since those bindings must be statically supplied by the programmer, rather than let the agent to dynamically perform this task. In this sense, the Semantic Web aims at offering machine-readable ontologies and reasoning tools to allow agents to autonomously find services whose exposed functionality fits agents' needs.

We have designed a Prolog-based reasoner implemented as a set of rules for computing semantic likeness between any pair of concepts from a shared ontology. This support is used by SWAM agents in order to determine the set of Web Services which best suit their semantic service requests. In the next subsections, we present our matchmaking support and reasoner.

3.1. Matching concepts

Ontologies are used to describe data and services in a machine-understandable way. In automated Web Service discovery systems, agents usually try to locate a sufficiently similar service to accomplish their current goal. The problem is indeed to define what "sufficiently similar" means. We propose to compute the similarity between a needed protocol and published services from the similarity, or degree of match, between individual concepts that describe their public properties and constituent elements, respectively. Therefore, the rest of this section describes how to determine the degree of match between any pair of concepts.

The degree of match between two concepts depends on their distance in a *taxonomy tree*. A taxonomy may refer to either a hierarchical classification of things, or the principles underlying the classification. Almost anything (objects, places, events, etc.) can be classified according to some taxonomic scheme. Anatomically, a taxonomy is a tree-like structure that categorizes a given set of objects. Like (Paolucci et al., 2002), we define four degrees of similarity between two concepts X and Y as follows:

- **exact** if X and Y are individuals belonging to the same or equivalent classes.
- **subsumes** if X is a subclass of Y.
- **plug-in** if Y is a subclass of X.
- **fail** occurs when none of the previous degrees apply.

Definition 1 states the degree of similarity in terms of OWL operators.

$$\textit{similarity degree}(X,Y) = \begin{cases} \textit{exact} & \text{if } X \text{ equivalentClass } Y, \\ \textit{subsumes} & \text{if } X \text{ subClassOf } Y, \\ \textit{plug-in} & \text{if } Y \text{ subClassOf } X, \\ \textit{fail} & \text{otherwise} \end{cases} \tag{1}$$

Although this scheme is very general and can be used to compute similarity between any pair of concepts, it has a drawback, which we will explain through an example. Fig. 2 illustrates how to use this similarity scheme for measuring the degree of match of four concept instances. In the figure, the names of the concepts have been chosen to denote their belonging classes. For example, concepts *a1* and *b1* are instances of classes *A* and *B*, respectively. Since *C* is a subclass of *B*, in the left side of Fig. 2 we labeled the similarity between *c2* and *b1* as "plug-in". Moreover, we labeled the similarity between *c2* and *a1* as "plug-in", because *C* (indirectly) subclasses *A* as well. Note that though *c2* is intuitively more similar to *b1* than to *a1*, this cannot be derived by employing the similarity scheme of Definition 1. Thus, we have enhanced this scheme by taking into account the distance between any pair of concepts in the taxonomy tree, as shown in the right side of the figure. Accordingly, the new similarity labels between *c1* and *b1* is "plug-in, 1", and "plug-in, 2" for *c1* with *a1*. Therefore, although these similarity labels share a qualitative value, the labels state that *c2* is hierarchically closer – i.e. more similar – to *b1* than to *a1*.

We have implemented in Prolog an algorithm to compute these enhanced similarity scheme between two arbitrary pair of concepts. Roughly, the algorithm consists of Prolog rules for computing the taxonomic distance between concepts. The recursive nature of Prolog fits very well for implementing this enhanced similarity scheme.

The rule match (X,Y,L,D) returns the distance D between a concept X and a concept Y under label L. In this sense, accepted values for L are "exact", "plug-in", "subsumes" and "fail", while D may take any positive integer as value. Note that having a hierarchical distance equals to zero means that the two concepts are equivalent, since they are at the same position in the taxonomy. The rule for computing the distance of equivalent classes is:

```
match(X, Y, exact, 0) : -equivalentClass(X, Y).
```

As we will show in next section, the antecedent of the rule is an appropriate Prolog translation of the OWL constructor "equivalentClass". Distance between a class and its directly related superclasses is defined as 1. Similarly, the distance between a class and its directly related subclasses is defined as 1. The rules associated with these two cases are:

```
match (X,Y,subsumes,1):- isSubClassOf (X,Y).
match (X,Y,plug-in,1):- isSubClassOf (Y,X).
```

Calculating the distance between indirect subclasses requires to vertically traverse the taxonomy tree and computing the

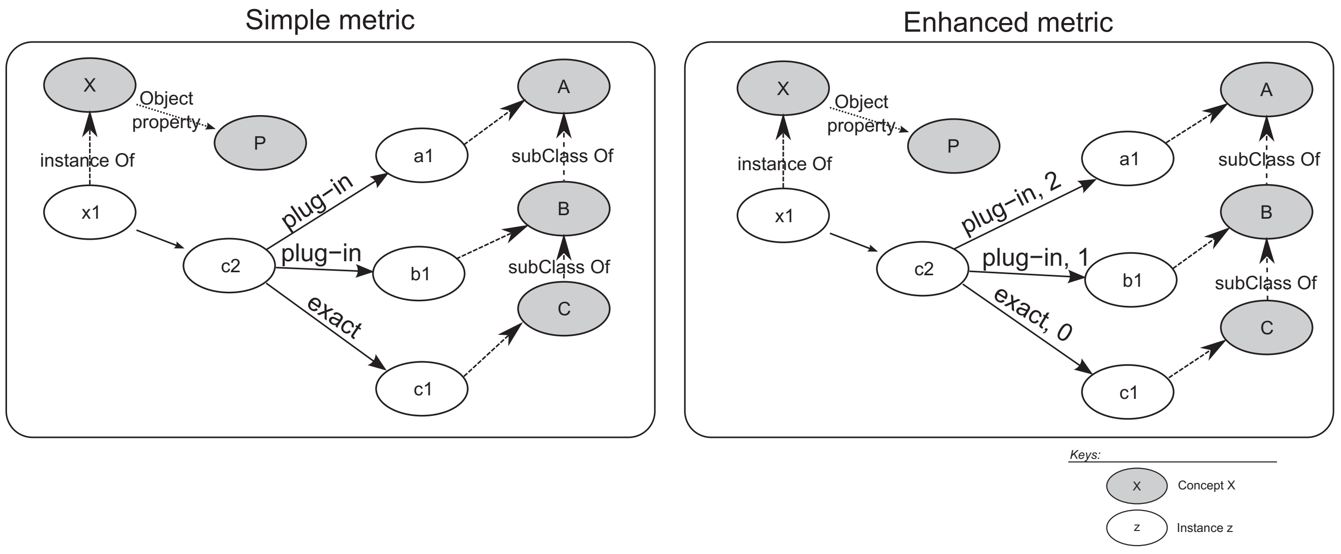


Fig. 2. Enhanced degree of match.

associated inheritance depth. Basically, distance for indirect subclassing is defined recursively by the following rules:

```
isSubClassOf (X,Y,1):- subClassOf (X,Y).
isSubClassOf (X,Y,D):- subClassOf (X,D), isSubClas-
sOf (Z,Y,D2), D is D2 + 1.
```

Finally, the matching rules for indirect subclasses are built on top of the rules for traversing the taxonomy:

```
match (X,Y,subsumes,D):- isSubClassOf (X,Y,D).
match (X,Y,plug-in,D):- isSubClassOf (Y,X,D).
```

For simplicity, the matchmaking support for concept properties is omitted. Nevertheless, it is implemented in a similar way to the scheme for classes previously discussed. The next subsection explains how we translate OWL constructors, such as "equivalentClass" or "subClassOf", onto Prolog.

3.2. Representing ontologies in Prolog

Our Prolog-based reasoner is built on top of the OWL-Lite language (Antoniou & van Harmelen, 2003). Interestingly, OWL-Lite is easily translatable to Prolog, since its semantics are equivalent to Description Logic, which is a decidable fragment of first-order logic (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2003). Table 1 shows the Prolog counterpart for some of the OWL-Lite sentences supported by our reasoner.

RDF triples are the core-blocks to express OWL-Lite constructors in Prolog. An RDF triple is a structure triple (subject, property, value) stating that *subject* is related via *property* to *value*. OWL-Lite features such as cardinality, range and domain constraints over properties are also translated to triples. For example, triple (author, domain, book) and triple (author, range, person) define that author can be applied to instances of class book with an instance of class person as value, while triple ('A Tale of Two Cities', author, 'Charles Dickens') states that *Charles Dickens* wrote a book named *A Tale of Two Cities.*

Most OWL-Lite features can be directly mapped as facts, which represent RDF triples. However, we build some rules for wrapping these facts, in order to improve the readability of the Prolog source, such as:

```
triple (X,'rdfs:subClassOf',Y):- subClassOf (X,Y).
triple  (X,'owl:equivalentClass',Y):- equivalent-
Class (X,Y).
```

Contrarily, there are two OWL-Lite features that cannot be directly mapped onto facts but require to use Prolog rules. OWL-Lite inequality and transitive sentences are supported through the following rules:

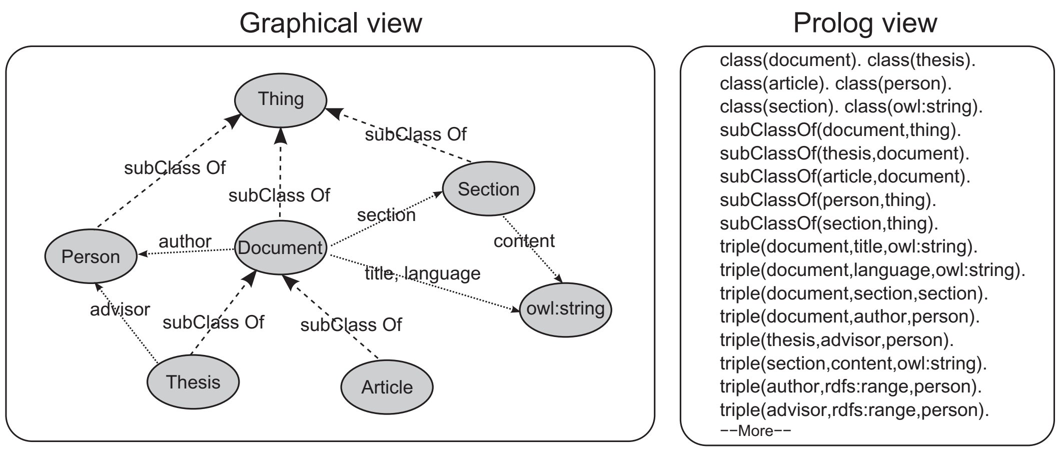
```
triple (Y,I,X):- inverseOf (P,I), triple (X,P,Y).
inverseOf (P,I):- triple (P, 'owl:inverseOf', I).
triple  (X,T,Z):- transitive  (T),  triple  (X,T,Y),
triple (Y,T,Z).
transitive  (T):- subClassOf  (T,  'owl:Transitive-
Property').
```

The first rule states that a concept Y is related to a concept X by property I whenever X is related to Y by a property P inverse to I. For example, if *wrote* and *author* were inverse properties, then triple ('Charles Dickens', wrote, 'A Tale of Two Cities') holds. The last rule handles transitive relationships between concepts. For example, if *taller_than* is a transitive property and *John* is taller than *Paul* and this latter is taller than *George*, then *John* is taller than *George*.

For exemplification purposes, the left side of Fig. 3 depicts a simple ontology for documents, while the right side of the figure illustrates its associated Prolog representation. The ontology

Table 1
OWL-Lite to Prolog correspondence.

OWL-Lite primitive	Prolog representation	Description
Class	Class (*X*)	*X* is a class
rdfs:subClassOf	subClassOf (*X*,*Y*)	*X* is a subclass of class *Y*
rdf:Property	property (*X*)	*X* is a property
rdfs:subPropertyOf	subPropertyOf (*X*,*Y*)	*X* is a subproperty of property *Y*
Individual	individualOf (*X*,*Y*)	*X* is an instance of class *Y*
inverseOf	inverseOf (*X*,*Y*)	*X* is inverse to property *Y*
equivalentProperty	equivalentProperty (*X*,*Y*)	*X* is equivalent to property *Y*
equivalentClass	equivalentClass (*X*,*Y*)	*X* is equivalent to class *Y*
Relationships	triple (*X*,*Y*,*Z*).	*X* is related to *Z* by property *Y*



```
class(document). class(thesis).
class(article). class(person).
class(section). class(owl:string).
subClassOf(document,thing).
subClassOf(thesis,document).
subClassOf(article,document).
subClassOf(person,thing).
subClassOf(section,thing).
triple(document,title,owl:string).
triple(document,language,owl:string).
triple(document,section,section).
triple(document,author,person).
triple(thesis,advisor,person).
triple(section,content,owl:string).
triple(author,rdfs:range,person).
triple(advisor,rdfs:range,person).
--More--
```

Fig. 3. An ontology for generic documents.

defines that *thesis* and *article* are both a *document* having one *author*. A thesis has also an *advisor*. Both *author* and *advisor* are properties whose values are instances of *person*. A document comprises a *title*, a *language* and *sections*. Finally, every section has *content*. In the previous rules, two new concepts appear: *Thing* and *owl:string*. Thing is the parent class of all OWL classes, since every class direct or indirectly inherits from it. Furthermore, OWL includes some built-in datatypes such as *owl:string*, *owl:long*, *owl:boolean*, to name a few, which allow to define literal properties.

3.3. Semantic Web service discovery

To perform a semantic search of Web Services instead of a less effective keyword-based search (e.g. by service name), an agent needs computer-interpretable descriptions about the functionality of services. Additionally, agents should semantically describe the services they need. Ontologies can be used for representing such descriptions. In this sense, OWL-S (Martin et al., 2007) is a collaborative effort which aims at creating a worldwide standard service ontology represented in OWL. OWL-S consist of a set of predefined classes and properties for representing services. OWL-S is intended to describe Web Services and how they must be invoked. On the other hand, SWAM allows agents to describe their service needs as a special kind of resources: *semanticWebServices*, as explained in Section 2.2. In order to manage OWL-S based descriptions of Web Services and match then onto SWAM protocols, we have built a semantic discovery infrastructure, which is shown in Fig. 4.

One of the main components of our semantic discovery system is its repository of shared ontologies. This repository stores the concepts used for semantically describing published Web Services. The current prototype of our service registry is based on an OWL-S sub-ontology named Service Profile, which offers support for semantic descriptions of functionality, arguments, preconditions and effects of Web Services. The repository of shared ontologies not only keeps previously used concepts but also allows publishers to reuse these concepts. In this way, a publisher can describe the functionality of his services and their input/output parameters in terms of concepts from this shared ontology repository. Instead, when none of the concepts of the shared ontologies database fits for describing a service, the publisher is allowed to add new ones. Upon publishing a Web Service, the service provider must also provide the URL of the corresponding WSDL document. We maintain a record that associates each WSDL document with the concepts involved in the semantic description of the corresponding Web Service in the Semantic Descriptions Database (SDD).

Discoverers, on the other hand, interact with our registry of Semantic Web Services by means of search requests. A search request is basically a semantic search expression given by a collection of ⟨property, expected_value⟩ pairs describing the desired conceptual value for some specific relationships within the Service Profile ontology. As we will explain later in this section, to rank candidate services we employ three properties from the Service Profile ontology: (1) operation functionality, (2) inputs and (3) outputs. Consequently, a search expression must partially or fully include the functionality, inputs and outputs that the discoverer expects to obtain from candidate services. These expressions, which may come from SWAM agents or conventional client applications, are appropriately translated to a Prolog query combining RDF triples and the match rule of Section 3.1. In general, such a Prolog query looks like:

```
triple (X, property, V), match (V, expected_value, L,
D).
```

with L and D representing the degree of match between V, i.e. the value of X for property, and the expected_value for that property. For example, the protocol of Section 2.2, which states that a Brainlet needs a Web Service for doing keyword-based searchers, generates the Prolog query:

```
?:-triple (Sx,‘owls:hasInput’,V0), match (V0,
‘s:KeywordList’,L0,D0),
tripe (Sx,‘owls:functionality’,V1), match (V1,
‘s:KeywordBasedSearch’,L1,D1).
```

where Sx represents the identifier of a service that has an input semantically similar to a *s:KeywordList* concept with a ⟨L0, D0⟩ degree of match, such as ⟨subsumes, 2⟩, and whose functionality matches into *s:KeywordBasedSearch*.

The semantic search engine (SSE) is the component of our registry in charge of processing and handling search requests, performing the discovery process and returning back the results. Processing a request means to translate it to a Prolog query. After processing a request, the SSE searches for Web Services that semantically matches[7] the requested conceptual functionality, inputs and outputs. To do this, the SSE compares the derived semantic query against the semantic descriptions of the published services. Likewise, the SEE exploits concept relationships, such as subclassing and equivalence, which have been also stored in the repository of shared ontologies. Then, a ranking algorithm is used to sort these

[7] A successful match between two concepts occurs when its similarity is greater than a application-given threshold.

results. The current materialization of the SSE has a default sort criterion which gives priority to the resulting degree of match between service functionality and outputs; however, this criterion can be changed dynamically. If two results have the same similarity, similarities are computed on their inputs to check that the requester is able to properly invoke any of the two services. Algorithm 1 describes the main steps of the proposed ranking algorithm.

Algorithm 1. Ranking algorithm

```
1:  procedure SORTRESULTS(result1,result2)
        ▷ Returns the first result
2:    if result1.operation.degreeOfMatch > result2.operation.degreeOfMatch then
3:      return result1
4:    end if
5:    if result1.operation.degreeOfMatch < result2.operation.degreeOfMatch then
6:      return result2
7:    end if
8:    if result1.output.degreeOfMatch > result2.output.degreeOfMatch then
9:      return result1
10:   end if
11:   if result1.output.degreeOfMatch < result2.output.degreeOfMatch then
12:     return result2
13:   end if
14:   if result1.input.degreeOfMatch > result2.input.degreeOfMatch then
15:     return result1
16:   end if
17:   if result1.input.degreeOfMatch < result2.input.degreeOfMatch then
18      return result2
19:   end if
20: end procedure
```

To better illustrate the notions exposed up to this point, the next section describes an example of a Semantic Web Service-enabled application coded with SWAM. We will put emphasis on how to implement SWAM agents that make use of protocols to semantically describe required Web Services.

4. A sample application

Let us suppose we are deploying a network composed of SWAM-enabled sites, where some of these sites offer Web Services for translating different types of documents (academic articles, forms, reports and so on) to a target language. Every time, a client wishes to translate a document, a Brainlet is asked to find the service that best adapts to the type of document being processed. In order to add semantics features to the model, all sites publish and search for Web Services by using SWAM, and services are annotated with concepts from the ontology presented in Section 3.2.

We also assume the existence of different candidate Web Services for handling the translation of a specific kind of document. For example, translating a plain document may differ from translating a thesis, since a more smart translation can be done in this latter case: a service can take advantage of a thesis' keywords to perform a context-aware translation. Nevertheless, note that a thesis could be also translated by a Web Service which expects a *document* concept as an input argument, as *thesis* concept specializes *document* according to our ontology.

When an agent receives a new document for translation, it prepares a semantic query. Here, the Brainlet is asked to translate a thesis to English. Fig. 5 shows the activities performed by the components involved in the translation process (the involved concepts are in italics). Before submitting the Web Service query, the Brainlet sets the desired service output as a *thesis*. Also, the Brainlet sets the target language as English and the source document type as *thesis*. Then, the semantic search process begins. SWAM its semantic matching capabilities to find all existing *translator* services. Let us suppose two services are obtained: a service for translating theses (S1) and a generic service (S2) for translating any type of document.

After finding a proper list of translation Web Services, SWAM sorts this list according to the degree of match computed between the semantic query and services descriptions and returns this new list back to the agent. In the example, the degree of match against S1 is greater than that of S2, because S1 outputs a *thesis* (exact matchmaking) while S2 results in subsumes matchmaking with distance one.

```
PROTOCOLS
  protocol (webService,[name (translate),in
   ([thesis,english]), out (thesis)],none).
CLAUSES
  % The Prolog structure representing some thesis
  thesis ([title (‘Title’),
    author (‘Author’),
    language (spanish),
    advisor (‘Advisor’),
    sections([...])).
  ?-translate (TargetLang, Res):-
  In = in ([thesis,TargetLang]),
  Out = out (thesis),
  webService([operation (translate), In, Out],
  WSProxy),
  thesis (Th),
  executeService (WSProxy, [Th, TargetLang], Res).
```

The above SWAM code implements the Brainlet discussed so far. When the webService(...) predicate of the CLAUSES section is reached, SWAM contacts its underlying semantic discovery subsystem to find candidate services that semantically match the Brainlet's request. The evaluation of the predicate returns a proxy (WSProxy) which is used to effectively invoke the resulting Web service. As explained before, the way this Web Service is actually contacted (i.e. migrate to the service location or invoke it remotely) is governed by access policies, in this case managed by the underlying platform as we have not configured a custom policy for accessing the service.

To sum up, the Brainlet has obtained a Web Service for execution using data semantic information rather than syntactic descriptions. To imagine a non-semantic matching scenario, assume that a syntactical categorization of services for translating documents is defined. Such a categorization would typically have a tree-like structure with a root node labeled "Document translator". The root would have two child nodes labeled "Article Translator" and "Thesis Translator", respectively. Without a semantic description about the type of documents each service is able to translate, the only way to find proper services is by their name, a pure syntactic and rigid mechanism. In this way, the logic to determine which service is appropriate for translating each type of document in terms of required inputs and delivered outputs remains hardcoded in the agent code. Furthermore, when a new type of document unknown to the agent is added, its implementation might need to be rewritten.

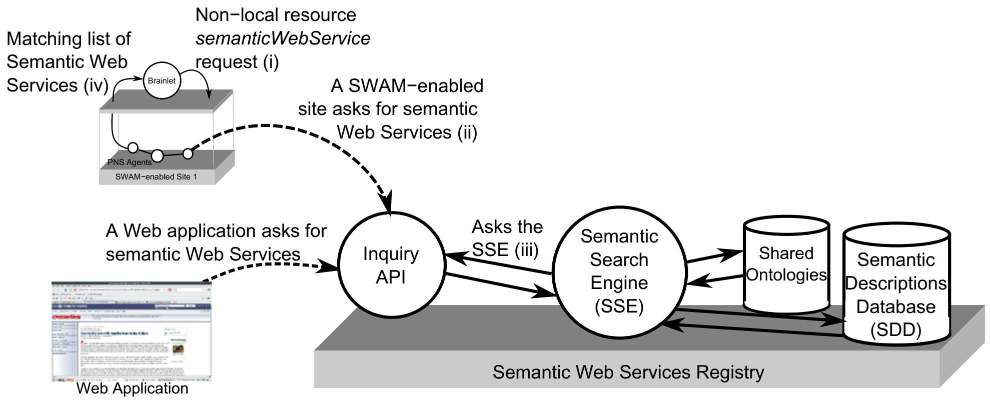

Matching list of Semantic Web Services (iv)
Non−local resource semanticWebService request (i)
Brainlet
A SWAM−enabled site asks for semantic Web Services (ii)
PNS Agents
SWAM−enabled Site 1
A Web application asks for semantic Web Services
Web Application
Inquiry API
Asks the SSE (iii)
Semantic Search Engine (SSE)
Shared Ontologies
Semantic Descriptions Database (SDD)
Semantic Web Services Registry

Fig. 4. SWAM and its Semantic Web Service discovery infrastructure.

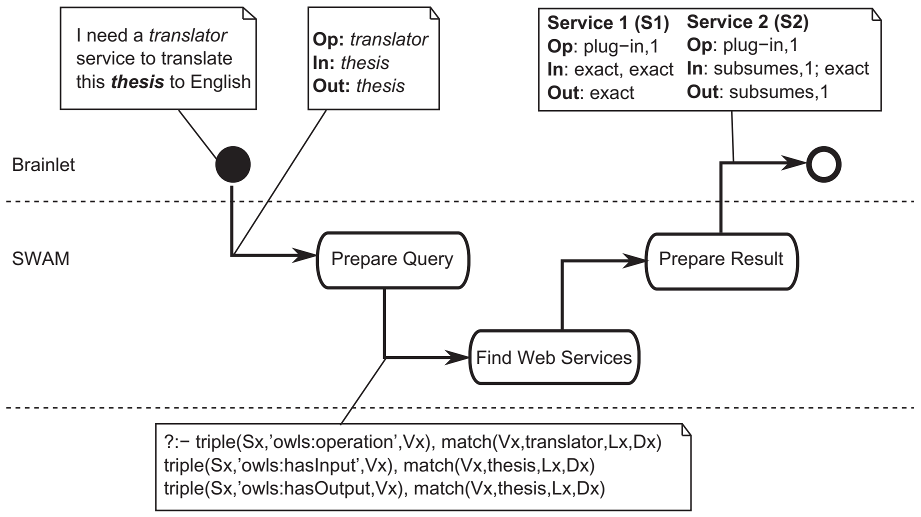

I need a translator service to translate this thesis to English
Op: translator
In: thesis
Out: thesis
Service 1 (S1)
Op: plug−in,1
In: exact, exact
Out: exact
Service 2 (S2)
Op: plug−in,1
In: subsumes,1; exact
Out: subsumes,1
Brainlet
SWAM
Prepare Query
Prepare Result
Find Web Services
?:− triple(Sx,'owls:operation',Vx), match(Vx,translator,Lx,Dx)
triple(Sx,'owls:hasInput',Vx), match(Vx,thesis,Lx,Dx)
triple(Sx,'owls:hasOutput',Vx), match(Vx,thesis,Lx,Dx)

Fig. 5. A Brainlet for thesis translation.

5. Experimental results

The next subsections report some experimental results obtained with a SWAM application for air ticket booking (without semantic matching of services) and several matchmaking simulations using its semantic discovery support. Basically, the first set of experiments assessed the performance of SWAM agents when interacting with Web Services, without considering semantics. On the other hand, the second set of experiments evaluated the performance of the semantic discovery support of SWAM.

5.1. An air ticket booking application

We developed several SWAM agents to solve a constraint problem in the air travel domain described in Martínez and Lespérance (2004). The goal of the application is to book an airplane ticket for traveling between two given cities according to certain domain business rules and user preferences. Examples of business rules are flight and seats availability, whereas examples of users' preferences are constrains over the ticket cost or air companies. We assumed the existence of Web Services for querying the availability of flights, seats, flight costs, and for booking a flight. The list of the operations offered by these Web Services are:

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 fulfilling the criteria.
2. checkSpace (Co, flightID): Checks whether "flightID" has available seats.
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.

In addition, we considered a fixed set of companies, known by the Brainlet implementing the application. For simplicity, the universe of constraints users may specify were grouped in five categories, which led to five different variants and hence implementations 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 BPF and BMxF.
- *Problem BBF (Book Best Flight):* This variant represents 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 one offered by a given preferred company.

We compared the performance of the SWAM solutions to the air ticket problem with equivalent implementations using IG-JADE-PKSLib (Martínez & Lespérance, 2004; Martínez, 2005), a toolkit for the development of multi-agent systems for Web Service composition and provisioning. IG-JADE-PKSLib is based on conventional AI planning techniques to perform service selection, which is a rather different approach to ours and therefore an interesting base for comparison.

The results of the IG-Jade-PKSLib implementations for the different variants of the flight booking problem are shown in Fig. 6. These results were extracted from Martínez and Lespérance (2004) and Martínez (2005). The experiments were performed on a XEON 3.0 GHz with 4 Gb RAM under Linux. The figure 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. In terms of performance, IG-JADE-PKSLib behaves reasonably well in BPF, BMxF and BPMxF, as service selections are performed in less than five seconds for five or less air companies. However, it can be seen from the figure that the implementation of these variants do not scale well for ten or more companies. Furthermore, BBF and BBPF were too slow in the tests, so we decided to left them out of the graphic.

Figs. 7 and 8 show the average execution times (five runs) of the SWAM implementations of each variant of the booking problem. In this case, the tests were conducted on a Pentium 4 2.2 GHz with

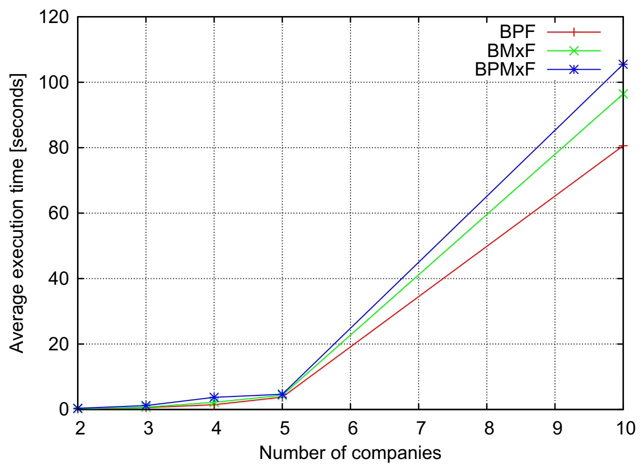


Fig. 6. Performance of BPF, BMxF and BPMxF (IG-Jade-PKSLib).

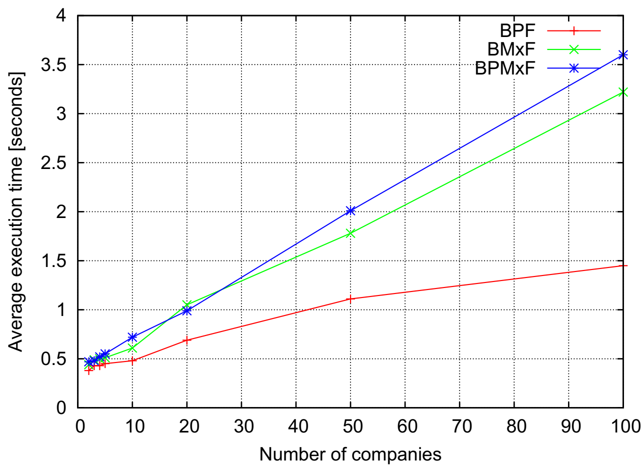


Fig. 7. Performance of BPF, BMxF and BPMxF (SWAM).

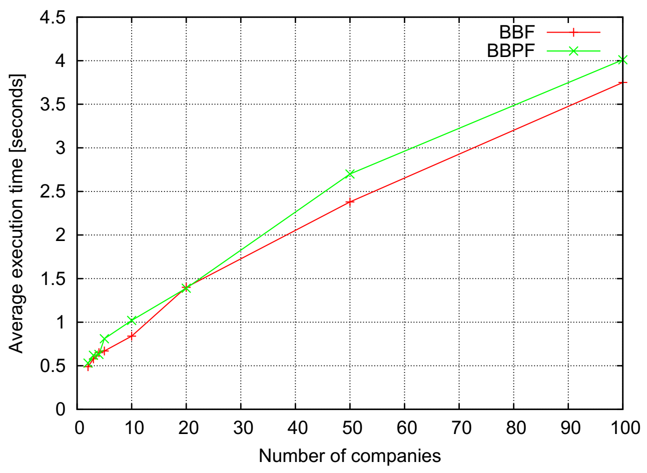


Fig. 8. Performance of BBF and BBPF (SWAM).

1 Gb RAM under Linux. The Web Services were deployed on the Apache Tomcat Web server, running on a second machine under Sun Java Virtual Machine (JVM) 1.5 (build 19).

As the reader can see, SWAM performed excellent, even when running on less powerful hardware. Unlike IG-JADE-PKSLib, all solutions (i.e. BPF, BMxF, BPMxF, BBF, BBPF) scaled well and performed in the order of few seconds. As expected, the worst execution times were obtained from BBF and BBPF variants, because they are the most computationally demanding problems. Specifically, the agent must find out the company that offers the least expensive ticket, which in turn requires checking prices in every company. All in all, these results are very encouraging since they suggest that using SWAM to simultaneously consume several Web Services does not lead to losing performance while maintaining both the benefits of Prolog for declaratively implementing mobile agent applications and the good features of GRMF and protocols for easily interacting with external Web Services.

5.2. Semantic matchmaking simulations

We evaluated the performance of our semantic approach to service discovery, in terms of response time, with regard to different sizes of the Semantic Description Database (SDD) component (i.e. the number of semantically annotated published Web Services). We developed a number of simple SWAM applications (without relying on mobility) that performed semantic search requests. Both the discovery infrastructure and all test applications were deployed on an Intel Pentium 4 working at 2.26 GHz and 512 MB of RAM, running Sun JVM 1.5 under Linux.

To fed the SDD, we first created two ontologies. The domains of these ontologies were stock management and car selling, respectively. Afterward, based on these two ontologies, we automatically created service descriptions and supplied them to the SDD. Each service description consisted of three properties: input, output and functionality. For example, the concepts involved in a service providing a quote for a sport car are *cs:sportcar* = input, *cs:quote* = output and *cs:car_quoting* = functionality ("cs" is the abbreviation for the "car selling" namespace).

Then, we published 100, 1000 and 10,000 semantic service descriptions. For each variant given by the three SDD sizes, we separately performed 200 search requests and took the elapsed times. Searches were simulated using randomly generated conditions and expected results. As mentioned above, searches were performed locally, this is, both the test applications and our registry of Semantic Web Services were deployed on the same machine. This was

Table 3
Performance of our semantic registry: Summary.

Number of published Web Services	100	1000	10,000
Average response time (ms)	2.37	12.65	149.33

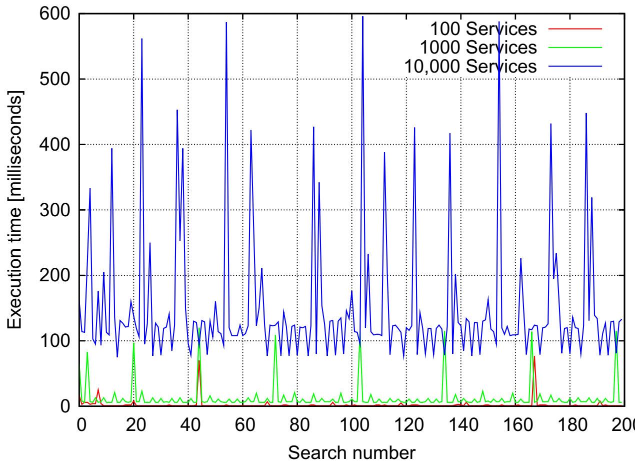


Fig. 9. Performance of our semantic registry: average search time.

done to avoid the noise introduced by network communication so as to accurately evaluate the retrieval performance of the discovery system.

Table 3 summarizes the resulting average response time for 600 random searches (i.e. 200 per test case). From the table, we can observe that the average performance of our discovery infrastructure was below 150 ms, even with 10,000 Web Services stored in the SDD. Graphically, Fig. 9 shows the relation between the size of the SDD and the search time. It is worth mentioning that the peaks of the curves are a consequence of overhead introduced by the JVM garbage collector. Nevertheless, the upper bounds of the response times are excellent regardless the size of the database.

6. Related work

There is an interesting body of efforts that has been being done on agent toolkits for leveraging Web Services. Some of the most relevant of such toolkits to our work are ConGolog (McIlraith & Son, 2002), IG-JADE-PKSLib (Martínez & Lespérance, 2004; Martínez, 2005) and MWS (Ishikawa, Tahara, Yoshioka, & Honiden, 2005; Ishikawa, Yoshioka, & Honiden, 2005). Despite some interesting advances towards the integration of agents and Web Services have been made, current proposals have the following problems: bad performance/scalability (IG-JADE-PKSLib), no/limited mobility (IG-JADE-PKSLib, ConGolog) and lack of support for common agent requirements such as knowledge representation, reasoning and high-level communication (MWS). Furthermore, none of the previous platforms provide support for semantic matching and discovery of Web Services. Another work close to ours is SmartResource (Katasonov & Terziyan, 2007), a platform for programming service-oriented multi-agent systems that is implemented on top of the JADE (Bellifemine, Caire, Poggi, & Rimassa, 2008) agent platform. Mobile agents are implemented in an XML-based language that extends RDF called S-APL (Semantic Agent Programming Language). This, however, makes SmartResource more difficult to adopt since logic-based languages (e.g. Prolog) are commonplace in agent programming.

Furthermore, there are a number of mobile agent-based platforms that address the problem of service access in wireless environments. Concretely, Baousis, Spiliopoulos, Zavitsanos, Hadjiefthymiades, and Merakos (2008) follows a framework-based approach to integrate mobile agents and Semantic Web Services, and therefore forces developers to have expertise on the framework before exploiting its capabilities. Similar to our agents, the structure of a mobile agent comprises code, data and migratory/cloning policies, plus some extra elements (e.g. an embedded semantic matching engine) that potentially makes them more heavier in terms of network resource usage than SWAM agents. Moreover, Terziyan (2005) and Adaçal and Benner (2006) follow the idea of using mobile agents to enable for mobile Web Services. Contrarily, we tackle the problem of simplifying the development of mobile applications and their interaction with stationary Semantic Web Services. Unfortunately, Adaçal and Benner (2006) does not handle service semantics, while the soundness of the approach proposed in Terziyan (2005) has not been corroborated experimentally yet.

Also, there are some proposals for semantic matching, publication and discovery of Web Services (Chiat, Huang, & Xie, 2004; Horrocks & Patel-Schneider, 2004; Sivashanmugam, Verma, Sheth, & Miller, 2003). One major limitation of these approaches is that their matching schemes do not take into account the distance between concepts within a taxonomy tree. As a consequence, similarity related to different specializations of the same concept are wrongfully computed as being equal. The most relevant work to the service matching approach of SWAM is the OWL-S Matchmaker (Kawamura, Hasegawa, Ohsuga, Paolucci, & Sycara, 2005), an UDDI-compliant semantic discovery and publication system. The OWL-S Matchmaker includes a semantic matching algorithm that is based on service functionality and data transformation descriptions which are made in terms of service input and output arguments. However, the OWL-S Matchmaker does not support taxonomic distance between concepts either.

Another interesting approach to semantic discovery of Web Services is proposed in Li and Horrocks (2004). Here, a Web Service is described using a OWL-S profile or by employing an extension of an existing one. Semantic similarity between two given services is therefore computed by comparing their associated profile metadata rather than matchmaking their input and output concepts. A service request must contain the class associated with the *ideal* service profile (i.e. the one preferred by the requester), which is matched against published profiles. The drawback of this approach is that it may be a cumbersome task for discoverers to build a service profile extension that properly describes their needs.

Finally, Bener, Ozadali, and Ilhan (2009) is another contemporary approach to discovery of Semantic Web Services. Functionally, the underlying matchmaking algorithm proposed in this work is very similar to ours. Particularly, the algorithm exploits the metamodel for Web Services defined by the OWL-S Service Profile and introduces a similarity scheme that considers taxonomic distance. However, the most significant difference between (Bener et al., 2009) and our approach to Semantic Web Service discovery is the idea of sharing ontologies. Specifically, Bener et al. (2009) incorporates the notion of independent ontologies, whereas we encourage sharing ontologies among service publishers.

The scheme adopted by Bener et al. (2009) allows publishers to describe their services using new concepts upon publishing a service, instead of encouraging them to inspect those concepts that were used in the past and, in turn, reuse or extend them, as our approach does. Then, many inconsistencies may spring, e.g. two services that produce the same output, but the former is described using *cs:Car* and the latter by means of *cs2:Automobile*. In this example, the outputs of the services will be treated as being different, unless their publishers explicitly indicate that *cs:Car* and *cs2:Automobile* are equivalent concepts. To mitigate this kind of inconsistency, Bener et al. (2009) also takes into account lexical relations from WordNet between the names of a pair of concepts. However, as the underpinnings of the WordNet (Fellbaum, 1989) -based approach to bridge different concepts lie in the syntax of concept names, names without any representative keywords may deteriorate the retrieval effectiveness of the proposed similarity scheme.

7. Conclusions

Intelligent mobile agents have the ability to infer, learn, act and move. Many researchers agree that they will play a key role in providing backbone technology to truly materialize the Semantic Web vision. In this light, our research aims at providing tools and platforms for easily building mobile agents and for allowing these agents to autonomously interact with Web Services. We have introduced SWAM, a language for programming Prolog-based mobile agents on the Semantic Web. A major difference between SWAM and other mobile agent toolkits is its support for reactive access to resources by failure, which reduces development effort by automatizing mobility and resource access decisions. In addition, its semantic matchmaking and discovery support helps agents to autonomously find and invoke Web Services. We have shown the practical usefulness of SWAM through a number of benchmark experiments. As explained, SWAM agents perform very well with respect to related approaches. Similarly, performance tests conducted on its discovery support gave promissory results.

Unlike previous work, SWAM defines a more precise semantic matchmaking algorithm, which is implemented on top of a Prolog-based reasoner that offers semantic inference capabilities over the decidable OWL-Lite ontology language. Our semantic infrastructure enables for the development of mobile agents that interact with Web-accessible functionality published across the WWW. This leads to the provisioning of an environment where every site can publish its capabilities as Semantic Web Services to which agents can find and access in a fully autonomous way.

There are some issues that are subject of future work. On one hand, we are developing more SWAM applications to assess its benefits from a software engineering perspective. For example, we are rebumping the Chronos (Zunino & Campo, 2009) mobile agent-based distributed meeting scheduler to exploit Web Services as well as semantic annotations. We are conducting research towards incorporating reasoning support for a more powerful and expressive language than OWL-Lite, such as OWL DL or OWL Full. However, this is rather challenging, as it is necessary to elaborate a convenient solution to address the "decidability versus expressive power" trade-off inherent to the sublanguages of OWL and to incorporate proper mapping rules into our Prolog-based reasoner or a bridge for querying a description logic reasoner. On the other hand, the Shared Ontologies database of our semantic registry must be extended to provide a framework to semantically describe, publish and discover other types of Web resources apart from Web Services, for example pages, blogs and other agents. Thereby, a software agent would be able to autonomously interact with Web Services or any kind of Web resource whose content and/or capabilities are defined in a machine-interpretable way. In addition, since the centralized nature of the discovery infrastructure may lead to scalability issues, we are extending the P2P facilities of GMAC (Gotthelf et al., 2008) with decentralized Semantic Web Service discovery.

References

Adaçal, M., & Benner, A. B. (2006). Mobile Web Services: A new agent-based framework. *IEEE Internet Computing, 10*(3), 58–65.

Antoniou, G., & van Harmelen, F. (2003). Web ontology language: OWL. In S. Staab & R. Studer (Eds.), *Handbook on ontologies in information systems. International handbooks on information systems* (pp. 67–92). Springer-Verlag.

Apache Software Foundation (2009). Apache CXF: An open source service framework. <http://cxf.apache.org>.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., & Patel-Schneider, P. F. (Eds.). . *The description logic handbook: Theory, implementation, and applications*. Cambridge University Press.

Baousis, V., Spiliopoulos, V., Zavitsanos, E., Hadjiefthymiades, S., & Merakos, L. (2008). Semantic Web Services and mobile agents integration for efficient mobile services. *International Journal on Semantic Web & Information Systems, 4*(1), 1–19.

Bellifemine, F., Caire, G., Poggi, A., & Rimassa, G. (2008). JADE: A software framework for developing multi-agent applications. Lessons learned. *Information and Software Technology, 50*(1–2), 10–21.

Bener, A., Ozadali, V., & Ilhan, E. (2009). Semantic matchmaker with precondition and effect matching using SWRL. *Expert Systems with Applications, 36*(5), 9371–9377.

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

Chiat, L. C., Huang, L., & Xie, J. (2004). Matchmaking for Semantic Web Services. In *IEEE international conference on services computing (SCC 2004)*. IEEE Computer Society.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., & Weerawarana, S. (2002). Unraveling the Web Services Web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing, 6*(2), 86–93.

Douglis, F. (2008). Ideas ahead of their time. *IEEE Internet Computing, 12*(5), 4–6.

Fellbaum, C. (1989). *WordNet: An electronic lexical database*. Bradford Books.

Fortino, G., Garro, A., & Russo, W. (2008). Achieving mobile agent systems interoperability through software layering. *Information and Software Technology, 50*(4), 322–341.

Gotthelf, P., Zunino, A., Mateos, C., & Campo, M. (2008). GMAC: An overlay multicast network for mobile agent platforms. *Journal of Parallel and Distributed Computing, 68*(8), 1081–1096.

Hendler, J. (2001). Agents and the Semantic Web. *IEEE Intelligent Systems, 16*(2), 30–36.

Horrocks, I., & Patel-Schneider, P. F. (2004). A proposal for an OWL rules language. In *13th international conference on World Wide Web (WWW 2004)*. New York, NY, USA: ACM Press.

Ishikawa, F., Tahara, Y., Yoshioka, N., & Honiden, S. (2005). A framework for synthesis of Web Services and mobile agents. *International Journal of Pervasive Computing and Communications, 1*(3), 227–245.

Ishikawa, F., Yoshioka, N., & Honiden, S. (2005). Mobile agent system for Web Service integration in pervasive network. *Systems and Computers in Japan, 36*(11), 34–48.

Katasonov, A., & Terziyan, V. (2007). SmartResource platform and semantic agent programming language (S-APL). In *Multiagent system technologies – 5th German conference (MATES 2007), Leipzig, Germany. Lecture notes in computer science* (Vol. 687, pp. 25–36). Berlin/Heidelberg: Springer.

Kawamura, T., Hasegawa, T., Ohsuga, A., Paolucci, M., & Sycara, K. (2005). Web Services lookup: A matchmaker experiment. *IT Professional, 07*(2), 36–41.

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

Leitner, P., Rosenberg, F., & Dustdar, S. (2009). Daios: Efficient dynamic Web Service invocation. *IEEE Internet Computing, 13*(3), 72–80.

Li, L., & Horrocks, I. (2004). A software framework for matchmaking based on Semantic Web technology. *International Journal of Electronic Commerce, 8*(4), 39–60.

Martin, D., Burstein, M., McDermott, D., McIlraith, S., Paolucci, M., Sycara, K., et al. (2007). Bringing semantics to Web Services with OWL-S. *World Wide Web, 10*(3), 243–277.

Martínez, E. (2005). Web Service composition as a planning task: An agent-oriented framework. Master's thesis, Department of Computer Science, York University.

Martínez, E., & Lespérance, Y. (2004). IG-JADE-PKSlib: An agent-based framework for advanced Web Service composition and provisioning. In *AAMAS-2004 workshop on web services and agent-based engineering*. New York, NY: Morgan Kaufmann Publishers.

Martínez, E., & Lespérance, Y. (2004). Web Service composition as a planning task: Experiments using knowledge-based planning. In *ICAPS-2004 workshop on planning and scheduling for web and grid services*. Whistler, British Columbia, Canada: Morgan Kaufmann Publishers.

Mateos, C., Crasso, M., Zunino, A., & Campo, M. (2006). Adding Semantic Web Services matching and discovery support to the MoviLog Platform. In M. Bramer (Ed.), *Artificial intelligence in theory and practice – IFIP 19th World Computer Congress, Santiago, Chile. IFIP international federation for information processing* (Vol. 217). Boston, MA, USA: Springer.

Mateos, C., Zunino, A., & Campo, M. (2007). Extending MoviLog for supporting Web Services. *Computer Languages, Systems & Structures, 33*(1), 11–31.

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

McIlraith, S. A., & Son, T. C. (2002). Adapting Golog for programming the Semantic Web. In D. Fensel, F. Giunchiglia, D. L. McGuinness, & M.-A. Williams (Eds.), *Proceedings of the 8th international conference on principles and knowledge representation and reasoning (KR-02), Toulouse, France*. New York, NY: Morgan Kaufmann.

Milanés, A., Rodriguez, N., & Schulze, B. (2008). State of the art in heterogeneous strong migration of computations. *Concurrency and Computation: Practice and Experience, 20*(13), 1485–1508.

OASIS Consortium (2004). UDDI version 3.0.2, UDDI Spec Technical Committee Draft. <http://uddi.org/pubs/uddi_v3.htm>.

Paolucci, M., Kawamura, T., Payne, T. R., & Sycara, K. P. (2002). Semantic matching of Web Services capabilities. In *First international semantic web conference on the semantic web*. Springer-Verlag.

Shadbolt, N., Berners-Lee, T., & Hall, W. (2006). The semantic web revisited. *IEEE Intelligent Systems, 21*(3), 96–101.

Sivashanmugam, K., Verma, K., Sheth, A. P., & Miller, J. A. (2003). Adding semantics to Web Services standards. In L. Zhang (Ed.), *IEEE international conference on web services (ICWS 2003)*. IEEE Computer Society.

Terziyan, V. (2005). Semantic Web Services for smart devices based on mobile agents. *International Journal of Intelligent Information Technologies, 1*(2), 43–55.

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

W3C Consortium (2004). RDF (Resource Description Framework). <http://www.w3.org/RDF>.

W3C Consortium (2007). SOAP version 1.2, W3C Candidate Recommendation. <http://www.w3.org/TR/soap>.

W3C Consortium (2007). WSDL version 2.0, part 1: Core language, W3C candidate recommendation. <http://www.w3.org/TR/wsdl20>.

Zunino, A., & Campo, M. (2009). Chronos: A multi-agent system for distributed automatic meeting scheduling. *Expert Systems with Applications, 36*(3, Pt. 2), 7011–7018.

Zunino, A., Mateos, C., & Campo, M. (2005). Enhancing agent mobility through resource access policies and mobility policies. In *V ENIA – XXV Congresso da Sociedade Brasileira de Computacão (SBC)*, San Leopoldo, RS, Brasil.

Zunino, A., Mateos, C., & Campo, M. (2005). Reactive mobility by failure: When fail means move. *Information Systems Frontiers – Special Issue on Mobile Computing and Communications, 7*(2), 141–154.