Adding Semantic Web Services Matching and Discovery Support to the MoviLog Platform

Cristian Mateos Marco Crasso Alejandro Zunino Marcelo Campo

ISISTAN Research Institute - Also CONICET
Facultad de Cs. Exactas, Departamento de Computación y Sistemas, UNICEN
Campus Universitario - Paraje Arroyo Seco - (B7001BBO) - Bs. As., Argentina
email: azunino@exa.unicen.edu.ar

Summary. Semantic Web services are self describing programs that can be searched, understood and used by other programs. Despite the advantages Semantic Web services provide, specially for building agent based systems, there is a need for mechanisms to enable agents to discover Semantic Web services. This paper describes an extension of the MoviLog agent platform for searching Web services taking into account their semantic descriptions. Preliminary experiments showing encouraging results are also reported.

1 Introduction

Once a big repository of Web pages, images and others forms of static data, the Web is evolving into a worldwide network of *Web Services*, paving the way to the so-called Semantic Web [1]. A Web Service [2] is a distributed piece of functionality that can be published, located and accessed through standard Web protocols. The goal of Web services is to achieve automatic interoperability between Web applications by providing them with an infrastructure to use Web-accessible resources.

Several researchers agree that mobile agents will have a fundamental role to materialize this vision [3, 4]. A mobile agent is a computer program which is able to migrate between network sites to perform tasks and interact with resources. Mobile agents have good properties that make them suitable for exploiting the potential of the Web [5]: support for disconnected operations, robustness and scalability.

Despite the advantages mobile agents offer, many challenges remain to glue them with Web services. Most of these challenges are a result of the nature of the Web. From its beginnings the Web has been mainly designed for human use and interpretation. Hence, mobile agents cannot autonomously take advantage of Web resources, thus forcing developers to write hand-coded solutions that are difficult to extend, reuse and maintain. Besides, the inherent complexity of mobile code programming with respect to traditional non-mobile systems, has dwindled the massive adoption of mobile agent technology, limiting its usage to small applications and prototypes.

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 agents for building distributed applications. To this end, we have developed MoviLog [6], a platform for building Prolog-based mobile agents on the WWW.

MoviLog encourages the usage of mobile agents by supporting a novel mechanism for handling mobility named RMF (Reactive Mobility by Failure). It allows programmers to easily build mobile agents on the Semantic Web without worrying about Web services location or access details. Furthermore, to take into account the semantics of services, we have extended MoviLog with support for semantic matching and discovery of Web services. The extension, called Apollo, enables an automatic interoperation between mobile agents and Web services with little development effort.

This paper is organized as follows. The next section introduces semantic Web services. Sect. 3 presents the MoviLog platform. Sect. 4 describes Apollo. Sect. 5 explains an example. Sect. 6 reports experimental results. Sect. 7 discusses the most relevant related work. Finally, Sect. 8 draws conclusions.

2 Semantic Web Services

Web services are a suitable model to allow systematic interactions of programs across the WWW. To hide the diversity of resources hosted by the WWW, Web services technologies mostly rely on XML, a structured language that extends and formalizes HTML. In this sense, the W3C Consortium has developed SOAP [1], a communication protocol based on XML. Besides, languages for describing Web services have been developed. An example is WSDL[2], an XML-based language for describing services as a set of operations over SOAP messages. From a WSDL document, a program can find out the specific services a Web site provides, and how to use and invoke these services.

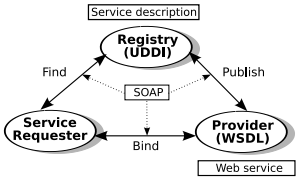

Fig. 1. Web services architecture

UDDI [3] defines mechanisms for searching and publishing Web services. By means of UDDI, Web service providers register information about the services they offer, thus making it available to potential clients. The information managed by UDDI ranges from WSDL files describing services to data for contacting providers.

Fig. 1 shows the conceptual architecture of Web services. A Web service is defined by a WSDL document describing a set of operations. A provider creates WSDLs for its services and publish them in an UDDI registry. A requester can browse registries to find services matching his needs. Then, the requesters can bind to the provider by invoking any of the operations defined by the WSDLs.

The weakest point of the architecture shown above is that it does not consider the semantics of services. To achieve an automatic interaction between agents and Web services, each service must be described in a nonambiguous and computer-understandable way. In this sense, some languages for Web services metadata annotation have emerged, such as RDF [4] and OWL [7], whose goal is to provide a formal

[1] SOAP (Simple Object Access Protocol): http://www.w3.org/TR/soap/
[2] WSDL (Web Service Description Language): http://www.w3.org/TR/wsdl
[3] UDDI (Universal Description, Discovery and Integration): http://www.uddi.org
[4] RDF (Resource Description Framework): http://www.w3.org/RDF/

model for describing the concepts involved in services. In this way, agents can *understand* and reason about the functionality a Web service performs, thus enabling the automatization of Web applications. Finally, a step towards the creation of a standard ontology of services is OWL-S [8]. The next section introduces MoviLog.

3 MoviLog

MoviLog [6] is a platform for programming mobile agents. The execution units of MoviLog are Prolog-based mobile agents named *Brainlets*. MoviLog uses strong mobility, where Brainlets execution state is transferred transparently on migration. Besides providing basic mobility primitives, the most interesting aspect of MoviLog is the notion of Reactive Mobility by Failure (RMF), a novel mobility model that reduces the effort for developing mobile agents by automating decisions such as when or where to migrate upon a *failure*. A failure is defined as the impossibility of an executing agent to obtain some required resource at the current site.

Roughly, each Brainlet possess Prolog code 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 RMF for managing mobility. RMF states that when a predicate declared in the protocols section of an agent fails, MoviLog moves the Brainlet along with its execution state to another site that contains definitions for the predicate. 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 RMF so that their failure may cause migration. The next example presents a simple Brainlet whose goal is to solve an SQL query given by a user on a certain database:

```
PROTOCOLS
  protocol(dataBase, [name(X), user(U), passwd(P)]).
CLAUSES
  doQuery(DBName, Query, Res):-
    dataBase([name(DBName), user('default'), passwd('')], Conn),
    doQuery(Conn, Query, Res), closeConnection(Conn).
  ?-sqlQuery(DBName, Query, Res):- doQuery(DBName, Query, Res).
```

PROTOCOLS section declares a protocol stating that the evaluation of *database(...)* predicate must be handled by RMF. In other words, the RMF mechanism will act whenever an attempt of connecting to the given database with the supplied username and password fails at the current site. As a result, RMF will transfer the agent to a site containing a database named *DBName*. After connecting to the database, the Brainlet will execute the query, and then return to its origin. Note that the protocol does not specify any particular value of the properties of the requested connection, which means that all unsuccessful attempts to access locally *any* database with *any* username-password combination will trigger reactive mobility.

Despite the advantages RMF has shown, it is not adequate for developing Web-enabled applications because it lacks support for interacting with Web resources. To overcome this limitation, RMF and its runtime support have been adapted to provide

a tight integration with Web Services [9]. Also, to take advantage of services semantics, an infrastructure for managing and reasoning about Web services metadata named Apollo has been built. The rest of the paper focuses on Apollo.

4 Semantic Matching in MoviLog

Semantic matching allows agents to take advantage of ontologies by using inference capabilities. An ontology represents the meaning of terms in vocabularies and the relationships between these terms [1]. Reasoners are often used to infer knowledge from ontologies. We have developed a Prolog-based reasoner as a set of rules and facts for describing and manipulating ontologies. In addition, the reasoner includes matchmaking rules to determine semantic similarity between any pair of concepts.

4.1 Representing ontologies in Prolog

We have developed a reasoner on top of the OWL-Lite language [7]. Unfortunately, OWL-Lite only supports classification hierarchy and simple constraints, thus offering less expressiveness than other languages belonging to the OWL family. However, OWL-Lite ensures inference completeness and decidability.

Table 1. OWL to Prolog correspondence

OWL-Lite	Prolog	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.
Properties	triple(X,Y,Z).	X is related to Z by property Y.

Interestingly, OWL-Lite can be translated to first order logic [10]. Table 1 shows the Prolog counterpart for some of the OWL-Lite sentences supported by our reasoner. OWL-Lite classes and properties are represented as simple facts; relationships are expressed as RDF triples. An RDF triple is a structure with the form *triple(subject, property, object)* which indicates that *subject* is related by *property* to *object* value. OWL-Lite features such as cardinality, range and domain constraints over properties are represented as triples. For example, *triple(author, range, person)* states that property *author* must be an instance of the class *person.* In addition, equality, inequality and transitive sentences of OWL-Lite may indirectly relate a concept to another. Our reasoner defines the following set of rules for dealing with these relationships:

```
triple(X,E,Y):- equivalentProperty(P,E), triple(X,P,Y).
triple(Y,O,X):- inverseOf(P,O), triple(X,P,Y).
triple(X,T,Z):- transitive(T), triple(X,T,Y), triple(Y,T,Z).
```

The first rule states that X is related to Y by property E, if E is equivalent to P and X is related to Y by P. For example, if *author* and *writer* were equivalent properties, then *triple(article, writer, person)* holds. The second rule states that Y is related to X by property O whenever *inverseOf(P,O)* is true and X is related to Y by P. For example, if *hasPublication* and *author* were inverse properties, then *triple(person, hasPublication, article)* holds. The last rule handles transitive relationships between concepts: if *John* is *Paul*'s *advisor*, and *Paul* is *George*'s *advisor,* then *John* is *George*'s *advisor.*

Fig. 2 shows an ontology for documents. It defines that a *thesis* and an *article* are *documents*, both having one or more authors. A *thesis* has an *advisor*. Both *author* and *advisor* are properties with range *person*. A document has a title, a language and some *sections*. Finally, a section has a content. In the rules two new concepts appear: *Thing* and *owl:string*. Thing is the parent class of all OWL classes. Also, OWL includes some built-in datatypes.

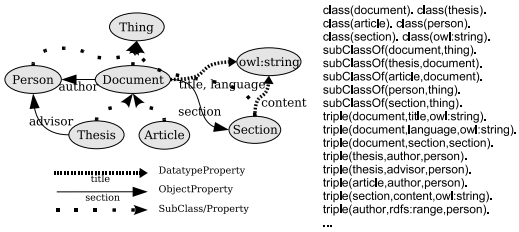

Fig. 2. An ontology for generic documents

4.2 Matching concepts

Ontologies can be used to describe data and services in a machine-understandable way. Automated data migration systems use ontologies to semantically describe their data structures. A process may then migrate a record from a source database to a sufficiently similar record in a target database. In automated Web services discovery systems, agents usually try to locate a sufficiently similar service to accomplish their current goal. Indeed, the problem to define what "sufficiently similar" means.

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 can be classified according to some taxonomic scheme. Mathematically, a taxonomy is a tree-like structure that categorizes a given set of objects. We have defined four degrees of matching according to [11]. The rational to compute the similarity between two concepts X and Y is:

- **exact** if X and Y are individuals belonging to the same or equivalent classes, we label similarity as **exact**.
- **subsumes** if X is a subclass of Y we label similarity as **subsumes**.
- **plug-in** if Y is a subclass of X we label similarity as **plug-in**.
- **fail** occurs when none of the previous labels could be stated.

We have enhanced this scheme by considering the distance between any pair of concepts in a taxonomy tree (see Fig. 3). From the diagram, it can be clearly stated

that *c2* is more similar to *b1* than *a1*: their similarity has been labeled as **plug-in**, but *c2* is hierarchically closer to *b1* than *a1*.

The matchmaking algorithm consists of a set of Prolog rules for calculating the distance between concepts within a taxonomy. The rule *match(C0, C1, Label, Dist)* returns the distance between *C0* and *C1* under *Label*. For example, the rule for equivalent classes is:

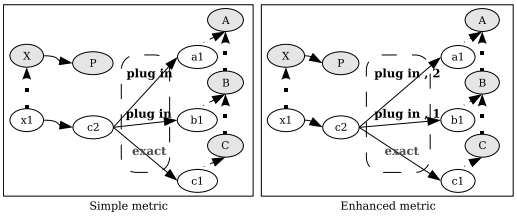

Fig. 3. Enhanced degree of match

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

The distance between two concepts is defined recursively as:

```
isSubClassOf(X,Y,1):- subClassOf(X,Y).
isSubClassOf(X,Y,N):- subClassOf(X,Z), isSubClassOf(Z,Y,T), N is T+1.
```

Applying the previous rules with X=article produces: *isSubClassOf(article,document,1)* and *isSubClassOf(article,thing,2)*. Matching rules for subsumes and plug-in labels use *isSubClassOf(X,Y,Z)* to compute distance as shown below:

```
match(X,Y,subsumes,N):- isSubClassOf(X,Y,N).
match(X,Y,plugin,N):- isSubClassOf(Y,X,N).
```

For space reasons, matchmaking support for properties is omitted. Nevertheless, the scheme previously discussed applies when computing distance between properties.

4.3 Semantic Web Services Discovery

In order to perform a semantic search of a Web service instead of a less effective keyword based search, an agent needs computer processable descriptions of services. Ontologies can be used for representing such descriptions. In this sense, OWL-S [8] aims at creating a standard service ontology. OWL-S consist of a set of predefined classes and properties for representing services. However, OWL-S is intended to describe services and how they must be invoked, but not how to semantically locate them. We combined OWL-S descriptions with UDDI registries to build a semantic Web services discovery system called Apollo. Fig. 4 shows its architecture.

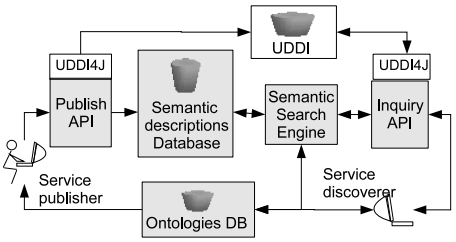

Fig. 4. The Apollo System

Apollo allows a Web service publisher to annotate services by using concepts from a shared OWL-S ontology database. Apollo is based on an OWL-S subset named Service Profile, which offers support for semantic description of services functionality, arguments, preconditions and effects. In this way, a publisher can describe services and its parameters in terms of concepts from the shared database. WSDL documents

are stored in UDDI nodes by using UDDI4J[5]. Finally, each WSDL document and its concepts are associated through the *Semantic Descriptions Database*.

A search request contains a concept describing the desired service functionality, and two sets of concepts for in/out parameters. To perform a more effective search, service requests are forwarded both to UDDI registries and to the Semantic Search Engine. Data resulting from an UDDI search are transformed to concepts from the *Ontology Database* by a component that extends the UDDI Inquiry API.

The main component of the *Semantic Search Engine* is the semantic reasoner. It uses a matchmaking scheme and a simple algorithm for sorting the results of a service search according to the degree of match. The algorithm first tries to contact a Web service that semantically matches the requested conceptual output. If there are more than one Web service with the same degree of match for their output, the algorithm examines inputs to check that the requester is able to invoke the service. The pseudo code for the Web service rating algorithm is:

```
exact = 2; subsumes = 1; plug_in= 0;
MatchResult compare(MatchResult mr0, MatchResult mr1) {
    if (mr0.output.label > mr1.output.label) return mr0;
    else if (mr0.output.label < mr1.output.label) return mr1;
    else { if (mr0.output.distance < mr1.output.distance) return mr0;
           else if (mr0.output.distance > mr1.output.distance) return mr1;
    }
    /* Outputs match... Now compare input parameters. */
}
```

5 A sample scenario

Suppose we are deploying a network composed of sites that accepts Brainlets for execution. Some of these sites offers Web services for translating different types of documents (articles, forms, theses, etc.) to a target language. Every time a client wishes to translate a document, an agent is asked to find the service that best adapts to the kind of document being processed. In order to add semantics features to the model, all sites publish and search for Web services by using Apollo, and services are annotated with concepts from the ontology presented in Sect. 4.1 (see Fig. 2).

We assume the existence of different instances of Web services for handling the translation of a specific type of document. For example, translating a plain document may differ from translating a thesis, because a smarter 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, since Thesis concept specializes Document according to our ontology.

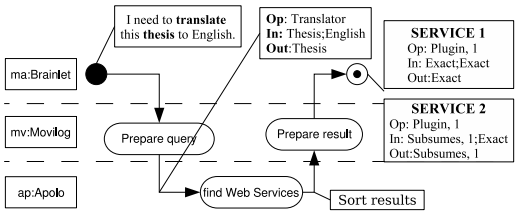

Fig. 5. A Brainlet for thesis translation

When a Brainlet gets a new document for translation, it prepares a semantic query. In this case, the agent needs to translate a thesis to English. Fig. 5 shows the activities

[5] UDDI for JAVA: http://www-124.ibm.com/developerworks/oss/uddi4j/

performed by each actor involved in the translation process. Before sending the service query, the Brainlet sets the service desired output as a Thesis. Also, the Brainlet sets the target language as english and the source document kind as Thesis, and then the semantic search process begins. Apollo uses semantic matching capabilities to find all existing Translation services. Let us suppose two services are obtained: a service for translating theses (*s1*) and a second service (*s2*) for translating any document.

After finding a proper list of translation Web services, Apollo sorts them according to the degree of match computed between the semantic query and services descriptions, and returns this new list back to the client. In the example, the degree of match for *s1* is greater than for *s2*, because *s1* outputs a Thesis (***exact***) while *s2* was labeled as ***subsumes*** with distance one.

```
PROTOCOLS
  protocol(webService, [ name(translate), input([thesis,english]), output(thesis) ]).
CLAUSES
  % The Prolog structure representing some thesis
  thesis([title('A title'), author('An author'), language(spanish),
          advisor('An advisor'), sections([...])]).
  ?-translate(TargetLang, Res):-
    webService([ name(translate), input([thesis,TargetLang]),
        output(thesis) ], WSProxy), thesis(Th), executeService(WSProxy,
        [Th, TargetLang], Res).
```

The previous code shows the implementation of the Brainlet discussed so far. As explained before, when the *webService(...)* predicate is executed, RMF contacts Apollo to find candidate services that semantically match the Brainlet's request. The evaluation of the predicate returns a proxy to the resulting service, which is used to effectively access it. The way the service is actually contacted (i.e. migrate to the service location or remotely invoke it) depends on access policies based on current execution conditions (network load, agent size, etc.) managed by the underlying platform.

6 Experimental results

In this section we report some experimental results. Particularly, we evaluated the performance of Apollo with regard to the number of published Web services. We generated a Semantic Web services database in an automatic fashion and we published it into Apollo. Both Apollo and all test applications were deployed on a Pentium 4 2.26 GHz with 512 MB of RAM, running Java 1.4.2 on Linux.

The Semantic Web services database was created by using two ontologies: a stock management domain and a car selling domain. Each service description was composed of five properties: input, output, category, preconditions and effects. Therefore, its input would be instantiated as a *cs:sportcar* concept, its output as a *cs:quote* concept, and finally its functionality as a *cs:car quoting* concept. Furthermore, another Web service can do the same for a "Sedan" car. In this case, since both *cs:sport* car and *cs:sedan* are *cs:vehicles*, service input would be instantiated as *cs:sedan*. Finally, searches have been simulated by using randomly generated conditions and expected results.

The resulting average response time for 600 random searches were: 2.37 ms (100 services), 12.65 ms (1000 services) and 149.33 ms. (10000 services). From this

we can conclude that Apollo performance is good. Note that the overall response time is less than 200 ms for 10000 Web services descriptions.

Fig. 6 shows the relationship between database size and the time for processing 200 different searches. It can be seen that the worst response time is less than 600 ms. Note that the peaks of the curves are caused by the JAVA garbage collector.

7 Related work

Some related approaches are [12, 13, 14]. Most of them describe services by means of ontologies and a discrete scale of semantic similarity based on [11]. One limitation of these approaches is that their matching scheme do not consider the distance between concepts within a taxonomy tree. Hence, similarity related to different specializations of the same concept are wrongfully computed as being equal.

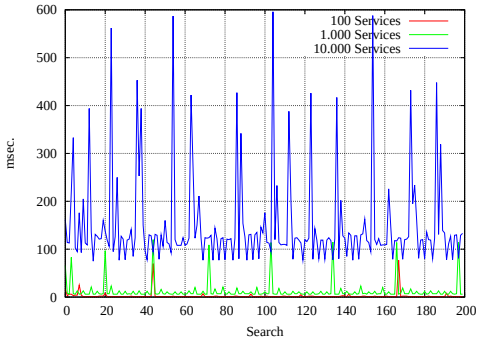

Fig. 6. # of searches vs. response time

The OWL-S Matchmaker [8] is a semantic Web service discovery and publication system. It includes a semantic matching algorithm based on service functionality and data transformation descriptions written in OWL-S. Data transformation descriptions are made in terms of service input and output arguments. Moreover, service search requests are enriched with concepts for describing the list of services that match a required data transformation. The OWL-S Matchmaker does not support taxonomic distance between concepts either.

In [15], a Web service is described by an OWL-S Service Profile instance or an extension of an existing profile. Semantic similarity between two services is computed by comparing their profiles' metadata instead of input/output concepts. A service request must contain the class associated to the *ideal* service profile (i.e. the one preferred by the requester), which is matched against published profiles. The drawback of this approach is its lack of support for finding available service profiles extensions.

Some interesting advances towards the integration of agents and Web services are ConGolog [16] and IG-JADE-PKSLib [17]. However, these approaches present the following problems: bad performance/scalability (IG-JADE-PKSLib), no/limited mobility (IG-JADE-PKSLib, ConGolog). In addition, none of the previous platforms provide support for semantic matching and discovery of Web services.

8 Conclusion and future work

This paper introduced Apollo, an infrastructure for semantic matching and discovery of Web services. Unlike previous work, Apollo defines a more precise semantic matching algorithm, implemented on top of a Prolog reasoner which offers inference capabilities

over OWL-Lite to a semantic Web services search engine. In addition, the integration of MoviLog with Apollo enables the development of mobile agents that interact with Web-accessible functionality. This leads to the creation of an environment where sites can publish their capabilities as Semantic Web services, so that agents can use them.

In the context of Apollo, some issues remain to be solved. First, OWL-Lite needs to be replaced by a more powerful and expressive language, such as OWL DL or OWL Full. Second, the Ontologies Database content must be enhanced in order to provide a framework to describe, publish and discover other types of semantically-annotated Web resources (pages, blogs or agents), and not just Web services. Thereby an agent would be able to autonomously interact with Web services or Web content.

References

1. T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic Web," *Scientific American*, vol. 284, pp. 34–43, May 2001.
2. S. J. Vaughan-Nichols, "Web services: Beyond the hype," *Computer*, vol. 35, Feb. 2002.
3. J. Hendler, "Agents and the semantic web," *IEEE Intelligent Systems*, vol. 16, Mar. 2001.
4. M. N. Huhns, "Software agents: The future of Web services," in *Agent Technology Workshops 2002*, vol. 2592 of *Lecture Notes in Artificial Intelligence*, pp. 1–18, 2003.
5. D. B. Lange and M. Oshima, "Seven good reasons for mobile agents," *Communications of the ACM*, vol. 42, pp. 88–89, Mar. 1999.
6. A. Zunino, C. Mateos, and M. Campo, "Reactive mobility by failure: When fail means move," *Information Systems Frontiers*, vol. 7, no. 2, pp. 141–154, 2005. ISSN 1387-3326.
7. G. Antoniou and F. van Harmelen, "Web Ontology Language: OWL," in *Handbook on Ontologies in Information Systems* (S. Staab and R. Studer, eds.), Springer-Verlag, 2003.
8. M. Paolucci and K. Sycara, "Autonomous semantic Web services," *IEEE Internet Computing*, vol. 7, no. 5, pp. 34–41, 2003.
9. C. Mateos, A. Zunino, and M. Campo, "Integrating intelligent mobile agents with Web services," *International Journal of Web Services Research*, vol. 2, no. 2, pp. 85–103, 2005.
10. J. de Bruijn, A. Polleres, and D. Fensel, "Deliverable D20v0.1 OWL Lite, WSML Working Draft." `http://www.wsmo.org/2004/d20/v0.1/`, June 2004.
11. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara, "Semantic matching of Web services capabilities," in *First International Semantic Web Conference*, vol. 2342, Springer, 2002.
12. K. Sivashanmugam, K. Verma, A. P. Sheth, and J. A. Miller, "Adding semantics to Web services standards," in *IEEE International Conference on Web Services*, June 2003.
13. I. Horrocks and P. F. Patel-Schneider, "A proposal for an owl rules language," in *The 13th international conference on World Wide Web*, pp. 723–731, ACM Press, Jan. 01 2004.
14. L. C. Chiat, L. Huang, and J. Xie, "Matchmaking for semantic Web services," in *IEEE International Conference on Services Computing (SCC'04)*, pp. 455–458, IEEE, 2004.
15. L. Li and I. Horrocks, "A software framework for matchmaking based on semantic Web technology," *International Journal of Electronic Commerce*, vol. 8, no. 4, pp. 39–60, 2004.
16. S. A. McIlraith and T. C. Son, "Adapting golog for programming the semantic Web," in *Fifth Symposium on Logical Formalizations of Commonsense Reasoning)*, May 20–22 2001.
17. E. Martínez and Y. Lespérance, "IG-JADE-PKSlib: An Agent-Based Framework for Advanced Web Service Composition and Provisioning," in *Workshop on Web Services and Agent-Based Engineering*, pp. 2–10, July 19–23 2004.