This is a preprint of the article: "C. Mateos, M. Crasso, A. Zunino and M. Campo: "Supporting Ontology-Based Semantic Matching of Web Services in MoviLog". Advances in Artificial Intelligence - Lecture Notes in Computer Science. Vol. 4140, pp. 390-399. Springer-Verlag. 2006. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/11874850_43

Supporting Ontology-Based Semantic Matching of Web Services in MoviLog

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

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

Abstract. The Web is moving towards the creation of a worldwide network of Web Services known as the Semantic Web. This new environment will allow agents to autonomously interact with Web information and services. This paper presents Apollo, an infrastructure which offers semantic matching and discovery capabilities to MoviLog, a platform for building mobile agents on the Semantic Web. Examples and experimental results showing the practical usefulness of Apollo are also reported.

1 Introduction

Once a big repository of pages and images, the Web is evolving into a worldwide network of *Web Services* 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 is to achieve automatic interoperability between applications by means of an infrastructure to use Web resources.

Several researchers agree that mobile agents will have a fundamental role in the materialization of this vision [3]. A mobile agent is a program able to migrate within a network to perform tasks or interact with resources. Mobile agents have suitable properties for exploiting the potential of the Web, such as support for disconnected operations, heterogeneous systems integration and scalability [4].

Despite these advantages, many challenges remain in order to glue mobile agents with Web Services technology. Most of them are a consequence of the nature of the WWW, since from its beginnings Web content has been designed for human interpretation [5]. Hence, unless content is described in a computer-understandable way, mobile agents cannot autonomously take advantage of the capabilities of Web resources, thus forcing developers to write hand-coded solutions that are difficult to reuse and maintain. This fact, together with the inherent complexity of mobile code programming, has affected the massive adoption of mobile agents and limited its use to small applications.

Indeed, there is a need for an agent infrastructure that addresses these problems and preserve the key benefits of mobile agent technology for building distributed applications. To cope with this, a platform for building logic-based mobile agents on the WWW named MoviLog [6] has been developed. MoviLog encourages the usage of mobile agents by supporting a mobility mechanism named RMF (Reactive Mobility by Failure). RMF allows programmers to easily code and deploy mobile agents on the

This is a preprint of the article: "C. Mateos, M. Crasso, A. Zunino and M. Campo: "Supporting Ontology-Based Semantic Matching of Web Services in MoviLog". Advances in Artificial Intelligence - Lecture Notes in Computer Science. Vol. 4140, pp. 390-399. Springer-Verlag. 2006. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/11874850_43

Semantic Web without worrying about Web Services location or access details. Furthermore, to consider the semantics of services, it provides an infrastructure for semantic matching and discovery of Web Services named Apollo, which allows for a truly automatic interoperation between mobile agents and services with little development effort.

The next section introduces semantic Web Services. Section 3 presents MoviLog. Section 4 describes Apollo. Section 5 explains an example. Section 6 reports experimental results. Section 7 surveys related work. Section 8 draws conclusions.

2 Semantic Web services

Web Services are a suitable model to allow systematic interactions of applications across the WWW. The Web Services model relies on XML, a structured language that formalizes HTML. In this sense, SOAP[1], a communication protocol based on XML, has been developed. In addition, languages for describing Web Services have been proposed. The most notorious example is WSDL[2], an XML-based language for describing services as a set of operations over SOAP messages. From a WSDL specification, a program can determine the specific services a Web site provides, and how to use and invoke these services.

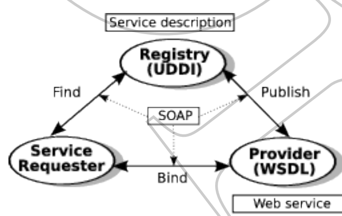


Fig. 1. Web Service conceptual architecture

UDDI[3] defines mechanisms for searching and publishing services written in WSDL. Providers register information about the services they offer, thus making it available to clients. The information managed by UDDI ranges from WSDL files describing services interfaces to data for contacting providers.

The most widely accepted architecture for Web Services is depicted in figure 1. A Web Service is defined as an WSDL interface describing a set of Web-accessible operations. The provider creates a WSDL document describing their Web Service and publishes it to a UDDI registry. A requester can browse registries to find a service that matches their needs along with the corresponding WSDL document. Finally, the requester can bind to the provider by invoking any of the operations defined in the WSDL document.

The weakest point of the architecture is that it does not consider the semantics of services. To achieve a truly automatic interaction between agents and Web Services, each service must be described in a computer-understandable way. Some languages for service metadata annotation have emerged, such as RDF[4] and OWL [7], whose goal is to provide a formal model for describing the concepts involved in services. In this way, agents can autonomously understand and reason about the precise functionality a Web Service performs, thus leading to a complete automatization of Web applications.

[1] SOAP Specification: http://www.w3.org/TR/soap/
[2] WSDL Specification: http://www.w3.org/TR/wsdl/
[3] UDDI: http://www.uddi.org
[4] RDF Specification: http://www.w3.org/RDF/

This is a preprint of the article: "C. Mateos, M. Crasso, A. Zunino and M. Campo: "Supporting Ontology-Based Semantic Matching of Web Services in MoviLog". Advances in Artificial Intelligence - Lecture Notes in Computer Science. Vol. 4140, pp. 390-399. Springer-Verlag. 2006. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/11874850_43

3 MoviLog

MoviLog [6] is a platform for programming mobile agents in the WWW. The execution units of MoviLog are Prolog-based strong mobile agents named *Brainlets*. Besides providing basic mobility primitives, the most interesting aspect of MoviLog is the notion of *reactive mobility* [8]. RMF is a mobility model that reduces agent developing effort by automating decisions such as when or where to migrate upon a *failure*. Conceptually, 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 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. 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 their failure may cause migration. The next example presents a simple Brainlet whose goal is to solve an SQL query given by the user on a certain database:

```
PROTOCOLS
 protocol(dataBase, [name(N), user(U), password(P)]).
CLAUSES
 doQuery(DBName, Query, Res):-
  ParamList=[name(DBName), user('default'), password('')],
  dataBase(ParamList, DBProxy),
  doQuery(DBProxy, Query, Res).
?-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 accessing a certain database fails in 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 to the properties of the requested resource (i.e. N, U and P variables), which means that all unsuccessful attempts to access locally *any* database with *any* username-password 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 has been adapted to provide a tight integration with Web Services [9]. Also, in order 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

Ontologies are used to explicitly represent the meaning of terms in vocabularies and the relationships between those terms [1]. In order to infer knowledge from ontologies,

This is a preprint of the article: "C. Mateos, M. Crasso, A. Zunino and M. Campo: "Supporting Ontology-Based Semantic Matching of Web Services in MoviLog". Advances in Artificial Intelligence - Lecture Notes in Computer Science. Vol. 4140, pp. 390-399. Springer-Verlag. 2006. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/11874850_43

Apollo includes a Prolog-based reasoner implemented as a set of rules to determine semantic similarity between any pair of concepts.

4.1 Representing ontologies in Prolog

The reasoner is built on top of the OWL-Lite language [7]. Interestingly, OWL-Lite is easily translatable to Prolog, since it has a Description Logic equivalent semantics, which are a decidable fragment of first-order logic [10].

Basic OWL-Lite constructors for classes and properties are represented as simple facts, while higher-level relationships are expressed as RDF *triples,* a structure *triple(subject,property,value)* stating that *subject* is related by *property* to *value.* OWL-Lite features such as cardinality, range and domain constraints over properties are also translated into triple. For example, *triple(author, range, person)* states that *author* must be an instance of class *person.* OWL-Lite equality, inequality and transitive sentences are also supported. For example, if *author* and *writer* were equivalent properties, then *triple(article,writer,person)* holds.

Figure 2 depicts a simple ontology for documents.

The ontology defines that both *thesis* and an *article* are *documents* having at least one *author*. A thesis has also an *advisor*. Both *author* and *advisor* are properties within range *person*. A document is composed of 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. Also, OWL includes some built-in datatypes such as *owl:string*, *owl:long*, *owl:boolean*, to name a few, which allow to define literal properties.

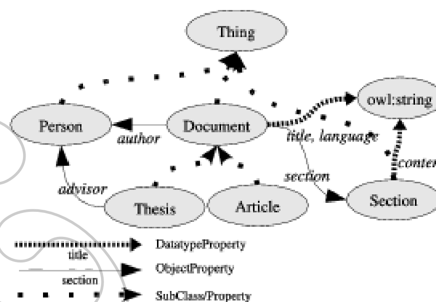


Fig. 2. An ontology for documents

4.2 Matching concepts

Ontologies are used to describe data and services in a machine-understandable way. In automated data migration systems, developers use ontologies to annotate semantically their data structures. A process may then be executed to 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 find a enough similar service to accomplish their current goal. The problem is indeed to define what “enough 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 (objects, places, events, etc.) can be classified according to some taxonomic scheme. Mathematically, a taxonomy is a

This is a preprint of the article: "C. Mateos, M. Crasso, A. Zunino and M. Campo: "Supporting Ontology-Based Semantic Matching of Web Services in MoviLog". Advances in Artificial Intelligence - Lecture Notes in Computer Science. Vol. 4140, pp. 390-399. Springer-Verlag. 2006. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/11874850_43

tree-like structure that categorizes a given set of objects. Like [11], Apollo defines four degrees of similarity between two concepts X and Y:

- **exact** if X and Y are individuals belonging to the same or equivalent classes.
- **subsumes** if X is a subclass of Y (for example *thesis* and *document*).
- **plug-in** if Y is a subclass of X (for example, *document* and *thesis*).
- **fail** occurs when none of the previous labels could be stated.

As shown in figure 3, this scheme is enhanced by taking into account the distance between any pair of concepts in a taxonomy tree. It can be clearly stated that *c2* is more similar to *b1* than to *a1*, and their similarity is labeled as **plug-in**.

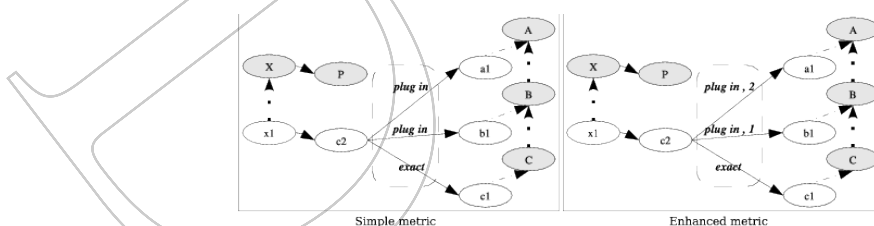


Fig. 3. Enhanced degree of match

The matchmaking algorithm consists of Prolog rules for measuring the taxonomic distance between concepts. The rule *match(C0, C1, L, D)* returns the distance D between concept C0 and concept C1 under label L. Distance between a class and its direct related superclasses is defined as one. Distance for indirect subclassing is defined recursively. For example, computing distance for C0=article and L=subsumes produces {C0=document, D=1} and {C1=thing, D=2}. Matchmaking support for properties works in a similar way.

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 (i.e. by service name), an agent needs computer-interpretable descriptions about the functionality of services. Ontologies can be used for representing such descriptions. OWL-S [12] is a worldwide effort which aims at creating a standard service ontology, describing services and how they must be invoked, but not how to semantically find them. Service Profile is an OWL-S subontology which offers support for semantic description of services functionality, arguments, preconditions and effects. The current prototype of Apollo combines Service Profile and UDDI to build a semantic Web Services discovery system.

Apollo allows a Web Service provider to publish its services to pre-configured UDDI registries, and to annotate these services by using concepts from a shared ontology database. Those annotations, which are made in terms of properties from the

This is a preprint of the article: "C. Mateos, M. Crasso, A. Zunino and M. Campo: "Supporting Ontology-Based Semantic Matching of Web Services in MoviLog". Advances in Artificial Intelligence - Lecture Notes in Computer Science. Vol. 4140, pp. 390-399. Springer-Verlag. 2006. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/11874850_43

Service Profile ontology, are stored onto a *Semantic Repository*. Finally, the semantic annotations and UDDI information (businessService and tModels) for a service are associated through a UDDI service key.

Apollo permits applications to search for Web Services both in UDDI nodes and in its Semantic Repository. A search request may contain both a semantic and a syntactic search condition, which are forwarded to the semantic reasoner and UDDI, respectively. The former is a collection of <*property, expected_value*> pairs, each one describing the desired conceptual value for some specific relationship *property* within the Service Profile ontology. For example, <*owls:hasOutput, thesis*> ask for services whose output is semantically similar to a thesis concept. On the other hand, a syntactic search condition is equivalent to a UDDI-like search request.

5 A sample scenario

Suppose a network comprising sites which accept Brainlets for execution. Some of these sites offers Web Services for translating different kind of documents 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 section 4.1.

It is assumed the existence of different instances of 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, since Thesis concept specializes Document according to our ontology.

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. 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. 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 this list according to the degree of match computed between the semantic query and services descriptions. In the example, the degree of match for *s1* is greater than for *s2*, because *s1* outputs a Thesis (***exact*** matchmaking) while *s2* matchmaking was labeled as ***subsumes*** with distance one.

```
PROTOCOLS
 protocol(service,[name(translate),in([thesis,english]),
                   out(thesis)]).
CLAUSES
 thesis([title('Title'), author('Author'), language(spanish),
```

This is a preprint of the article: "C. Mateos, M. Crasso, A. Zunino and M. Campo: "Supporting Ontology-Based Semantic Matching of Web Services in MoviLog". Advances in Artificial Intelligence - Lecture Notes in Computer Science. Vol. 4140, pp. 390-399. Springer-Verlag. 2006. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/11874850_43

```
        advisor('Advisor'), sections([...])}).
?-translate(TargetLang, Result):-
 Props=[name(translate),in([thesis,TargetLang]),out(thesis)]
 service(Props,WSProxy), thesis(Th),
 executeService(WSProxy, [Th, TargetLang], Result).
```

The previous code implements the Brainlet discussed so far. When the *webService(...)* predicate is executed, RMF contacts Apollo in order to find candidate services that semantically match the Brainlet's request. The evaluation of the predicate returns a proxy which is used to effectively access the service. The way the service is contacted (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.

To sum up, the Brainlet has obtained a Web Service for execution using data semantic information rather than a syntactic description. To imagine a nonsemantic matching scenario, assume that is defined a syntactical categorization of services for translating documents. Such a categorization will typically have a tree-like structure with a root node labeled "Document translator". The root will have two children nodes labeled "Article Translator" and "Thesis Translator", respectively. Without a semantic description about the kind of document 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 kind of document remains hard-coded in the agent. Also, when a new kind of document unknown to the agent is added, its implementation becomes obsolete.

6 Experimental results

6.1 Performance tests

Test cases were conducted with regard to various size of the description database. Both Apollo and all test applications were deployed on an Intel Pentium 4 working at 2.26 GHz and 512 MB of RAM, running Sun JVM 1.4.2 on Linux.

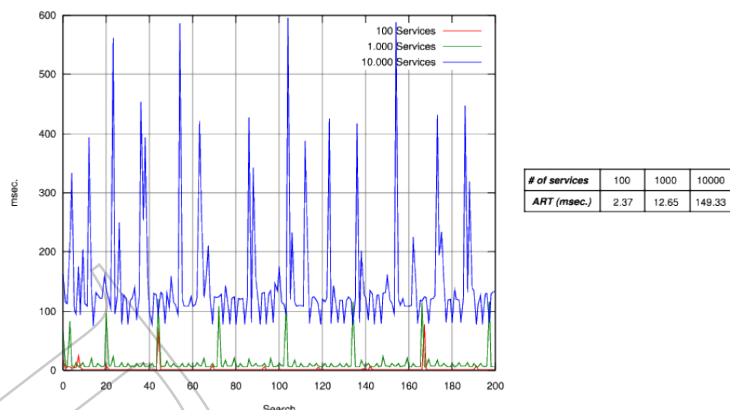
The database was automatically created based on two ontologies: stock management and car selling. Each service description was composed of five properties: input, output, category, preconditions and effects. For example, the concepts involved in a service providing a quote for a sport car are *cs:sportcar* (input), *cs:quote* (output) and *cs:car_quoting* (functionality). Searches were simulated by using randomly-generated conditions and expected results.

Figure 4 shows the relation between database size and the time for processing 200 different searches by test case. The worst response time is 600 ms., even with 10.000 Web Services. It is worth noting that the peaks of the curves are a consequence of overhead introduced by the JAVA garbage collector. Similarly, the table summarizes the resulting average response (ART) time for 600 random searches. It shows that Apollo performance was outstanding. The ART was less than 200 ms., even with 10.000 services stored in the database.

Draft

This is a preprint of the article: "C. Mateos, M. Crasso, A. Zunino and M. Campo: "Supporting Ontology-Based Semantic Matching of Web Services in MoviLog". Advances in Artificial Intelligence - Lecture Notes in Computer Science. Vol. 4140, pp. 390-399. Springer-Verlag. 2006. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/11874850_43



# of services	100	1000	10000
ART (msec.)	2.37	12.65	149.33

Fig. 4. searches vs. response time

6.2 Comparison against ASP

This subsection presents a brief comparison between the implementation of our reasoner and another implementation using ASP [13,14]. ASP is a logical programming language with a syntax similar to Prolog but, instead of a query driven interface, ASP offers a *result set* driven interface. A result set (or stable model) is a view of all rules that can simultaneously coexist. Given an ASP program and a query, the output is a stable model. ASP has been recently proposed as a formalism for providing inference capabilities for the Semantic Web.

Comparison between the two implementations showed that a result set driven interface is less adequate for semantic matchmaking, since it requires to analyze the result set in order to find the rule(s) storing the resulting semantic similarity. Notice that the output of any ASP program is usually a big list of those rules which comforms a stable model for a query where contradictory knowledge does not exist. This clearly adds an overhead when parsing the output of a query.

Both implementations represent OWL-Lite information as RDF triples, which is the only feature they have in common. Rules for computing taxonomic distance from Apollo's implementation are more concise and clear than ASP's. On the other hand, the performance of Apollo's reasoner is benefited from the query driven interface inherent to Prolog. The ASP reasoner produced large result sets even for simple queries and a description database of 100-1000 services (39 predicates against 1 for the case of Prolog).

7 Related work

There are some proposals for semantic matching, publication and discovery of Web services [15,16]. One major limitation of these approaches is that their matching scheme do not take into account the distance between concepts within a taxonomy tree. As a result, similarity related to different specializations of the same concept are wrongfully computed as being equal.

This is a preprint of the article: "C. Mateos, M. Crasso, A. Zunino and M. Campo: "Supporting Ontology-Based Semantic Matching of Web Services in MoviLog". Advances in Artificial Intelligence - Lecture Notes in Computer Science. Vol. 4140, pp. 390-399. Springer-Verlag. 2006. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/11874850_43

The most relevant work to our approach is the OWL-S Matchmaker [12], a semantic Web Service discovery and publication system that is UDDI-compliant. It 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. Service search requests are enriched with concepts describing the list of services that match a required data transformation. This approach does not support taxonomic distance between concepts either.

With respect to the plethora of work on mobile agents tools for the Semantic Web, some interesting advances are ConGolog [17], IG-JADE-PKSLib [18] and MWS [19]. However, these approaches present 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 and reasoning (MWS). Also, none of the previous platforms supports semantic matching and discovery of Web Services.

Finally, some modeling frameworks for creating Semantic Web enabled Web Services have been recently proposed, such as SWSA [20] and WSMO[5]. However, rather than specific components, these models offer abstract specifications for materializing infrastructure and ontologies for the Semantic Web.

8 Conclusion and future work

This paper introduced Apollo, an infrastructure which offers semantic matching and discovery capabilities, and described how this support has been integrated to a mobile agent platform named MoviLog.

Unlike previous work, Apollo defines a more precise matching algorithm, implemented on top of a Prolog reasoner which offers inference capabilities over OWL-Lite annotations. In addition, the integration of MoviLog with this support enables the development of mobile agents that interact with Web-accessible functionality published across the WWW. This leads to the creation of an environment where every site can publish its capabilities as Semantic Web Services to which Brainlets can find and access in a fully autonomous way.

Regarding Apollo, some issues remain to be solved. First, OWL Lite needs to be replaced by a more powerful and expressive language (e.g. OWL DL or OWL Full). Second, Ontologies Database content must be enhanced in order to provide a framework to describe, publish and discover any type of semantically annotated Web resource (pages, blogs, other agents), and not just Web Services. Thereby a software agent would be able to autonomously interact with any type of Web content defined in a machine-processable way.

References

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

[5] WSMO: http://www.wsmo.org

This is a preprint of the article: "C. Mateos, M. Crasso, A. Zunino and M. Campo: "Supporting Ontology-Based Semantic Matching of Web Services in MoviLog". Advances in Artificial Intelligence - Lecture Notes in Computer Science. Vol. 4140, pp. 390-399. Springer-Verlag. 2006. ISSN 0302-9743."

The original publication is available at http://dx.doi.org/10.1007/11874850_43

2. Vaughan-Nichols, S.: Web services: Beyond the hype. Computer **35**(2) (2002)
3. Hendler, J.: Agents and the semantic web. IEEE Intelligent Systems **16**(2) (2001)
4. Lange, D.B., Oshima, M.: Seven good reasons for mobile agents. Communications of the ACM **42**(3) (1999) 88–89
5. McIlraith, S., Son, T.C., Zeng, H.: Semantic web services. IEEE Intelligent Systems, Special Issue on the Semantic Web **16**(2) (2001) 46–53
6. Zunino, A., Campo, M., Mateos, C.: Simplifying mobile agent development through reactive mobility by failure. In: SBIA'02. Volume 2507 of Lecture Notes in Computer Science., Springer-Verlag (2002) 163–174
7. Antoniou, G., van Harmelen, F.: Web Ontology Language: OWL. In Staab, S., Studer, R., eds.: Handbook on Ontologies in Information Systems, Springer-Verlag (2003)
8. Zunino, A., Mateos, C., Campo, M.: Reactive mobility by failure: When fail means move. Information Systems Frontiers. Special Issue on Mobile Computing and Communications **7**(2) (2005) 141–154 ISSN 1387-3326.
9. Mateos, C., Zunino, A., Campo, M.: Extending movilog for supporting web services. Computer Languages, Systems & Structures (2006) To appear.
10. 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 (2003)
11. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.P.: Semantic matching of web services capabilities. In: ISWC'02. Volume 2342., Springer-Verlag (2002)
12. Paolucci, M., Sycara, K.: Autonomous semantic web services. IEEE Internet Computing **7**(5) (2003) 34–41
13. Marchi, M., Mileo, A., Provetti, A.: Declarative policies for web service selection. In: POLICY'05, Sweden, IEEE Computer Society (2005) 239–242
14. Ruffolo, M., Leone, N., Manna, M., Saccà, D., Zavatto, A.: Exploiting asp for semantic information extraction. In: ASP'05. Volume 142 of CEUR Workshop Proceedings. (2005)
15. Horrocks, I., Patel-Schneider, P.F.: A proposal for an owl rules language. In: WWW'04, ACM Press (2004) 723–731
16. Chiat, L.C., Huang, L., Xie, J.: Matchmaking for semantic web services. In: SCC'04, IEEE Computer Society (2004) 455–458
17. McIlraith, S.A., Son, T.C.: Adapting golog for programming the semantic web. In: Common sense'01. (2001)
18. Martínez, E., Lespérance, Y.: IG-JADE-PKSlib: An Agent-Based Framework for Advanced Web Service Composition and Provisioning. In: WSABE'04. (2004) 2–10
19. Ishikawa, F., Yoshioka, N., Tahara, Y., Honiden, S.: Towards synthesis of web services and mobile agents. In: WSABE'04. (2004)
20. Burstein, M., Bussler, C., Zaremba, M., Finin, T., Huhns, M.N., Paolucci, M., Sheth, A.P., Williams, S.: A semantic web services architecture. IEEE Internet Computing **9**(5) (2005) 72–81