Semantic-Based Web Mining Under the Framework of Agent

Usha Venna

K Syama Sundara Rao

Abstract— To make automatic service discovery possible, we need to add semantics to the Web service. A semantic-based Web mining is mentioned by many people in order to improve Web service levels and address the existing Web services requested by the people. The backbone of this solution is clearly the UDDI (private) registry. Earlier for web mining service they use WSDL-S approach, which had undergone many semantic problems. Since WSDL-S is a light weight solution approach it fails in reaching the efficiency levels of web mining service. To overcome this issue I am proposing a new solution by using OWL-S upper ontologies, which is a full solution for achieving an efficient web mining service. A matching algorithm is designed in OWL-S approach which specifies the semantic matching between a service request and a service description which does Semantic-based Web data mining by combining the semantic Web and Web mining. Software that implements a given matching algorithm is called a matchmaking engine. Practical implementation of this OWL-S approach in Semantic Web makes Web mining easier to achieve, but also can improve the effectiveness of Web mining. Here I am giving knowledge about semantic web and web mining. Finally I propose to build a semantic-based Web mining model under the framework of the Agent.

Keywords— Semantic web, Semantic based web data mining, Comparison.

I. INTRODUCTION

Recently, public UDDI registries have been shut down by major players such as Microsoft, IBM, and SAP. This may or may not be a problem for us. Private UDDI registries continue to exist and are used by different organizations; semantically enhanced UDDI registries can still be built within the organization and used for automatic discovery of requested services. However, if the goal is to automatically search for a service over the Internet (certainly outside the domain of the organization), then it is just not possible; we would have to find another public registry to hold all the Semantic Web service descriptions. This could dramatically change the whole work flow. More precisely, if the public UDDI registries were not shut down, a service provider could continue using some UDDI tools to map the semantic description of a given web service into UDDI entries. Also, a service requester could have used UDDI APIs or some other embedded matchmaking engine to automatically search for a requested service.
Now, with the shutdown of UDDI public registries, the foregoing workflow is not feasible anymore.
It is, however, not wise to follow the path of UDDI by creating

yet another centralized repository and asking the potential service providers to publish their semantic service descriptions into this newly created registry.

A possible solution, however, is to take the burden away from both the service publishers and service requesters by making the whole publish- search cycle more transparent to all these users. More specifically, this can be done as follows:

- Service providers could simply publish their services using OWL-S or WSDL-S (or any other markup language) on their own hosting Web sites, without worrying about using APIs or mapping tools to publish the services into some centralized repository.
- A Web crawler could collect all these semantic descriptions from their individual publishing sites and save them into a centralized repository, without the providers' knowledge; only this crawler would have knowledge about the specific structure of the registry.
- Service requesters could form and submit their service requests to the central registry; again, they should ensure the service requests are expressed using OWL- S or WSDL-S (or any other markup language); no knowledge about the structure of the registry is needed.

- The centralized registry assumes the responsibility of receiving the service requests, invoking some matchmaking engine, and returning a candidate set to the service requesters.

Such a solution will make it possible for a service requester to look for a desired service without even knowing whether it exists; the only requirement is to express the request in a semantic markup language such as OWL-S. Also, this solution will make it possible for a service provider to publish service descriptions directly on his or her Web server without even knowing how the service descriptions are collected and discovered; again, the service descriptions should be constructed using a markup language such as OWL-S.

This solution certainly does not need any public UDDI as its backbone support; it works almost like a search engine, except that it is a specialized search engine: a search engine for the automatic discovery of Semantic Web services.

We are going to design this search engine in detail, and we are also going to do some coding to implement it as a prototyping system to show how the added semantics can help a service requester find the required services automatically. We have reached the last part of this book, and we hope that the design of such a search engine and all the related coding exercises will help you to put together all that you have learned so far and will give you a more concrete and detailed understanding of the Semantic Web and Semantic Web services.

II. DESIGN OF SEARCH ENGINE

A. Architecture of Search Engine

The design and the functionality of each component and the interaction among the components will be discussed now.

B. Individual Components

The first component for discussion is the Web crawler. The goal of the crawler is to collect the Semantic Web service descriptions; this is done by visiting the Web, finding the hosting Web sites, and parsing the service descriptions. These descriptions can be written in OWL-S or WSDL-S, or in any other markup language. Clearly, if the service publisher does not offer a Semantic Web service description, the published Web service will not be collected into the Semantic Web Service Repository.

This crawler is an example of a *focused* crawler:

It does not just collect everything it encounters; instead, it collects only the Semantic Web service descriptions. (We will see how the crawler decides whether a given page is such a description or not in the implementation section.) An obvious problem that needs to be considered is the "sparseness" of the service descriptions. In other words, there are not many Semantic Web service descriptions on the Internet for collection, and most pages the crawler visits would have nothing to do with Semantic Web services. Therefore, precious time and system resources could be wasted.

A possible solution to this problem is to come up with a better set of seed URLs. For instance, we can use Swoogle to search for all the documents that use the OWLS upper ontology; the results returned by Swoogle are then used as an initial set of seed URLs. You can also design other heuristics to handle this problem.

The next component is the Semantic Web service repository. Its main purpose is to store the descriptions of Semantic Web services. It is similar to the index database in a traditional search engine. The structure of the repository is designed to provide enough information for the discovery process. Our current design of the repository contains two main tables. The first table is called mainRegistry, and its structure is shown in Table 1

TABLE 1

Structure of the mainRegistry Table

Field Name	Meaning
serviceID	Key to identifying this service; the URL of the Semantic Web service description file is used as the value of this field
ontologyURL	Each Semantic Web service description has to be constructed based on some ontology; this is the URL of the ontology document
serviceName	Name of this service
contactInfo	Contact information of this service; it could be an e-mail address or phone number, fax number, etc.
WSDLURL	URL address of the WSDL document for this Web service

TABLE 2
Structure of the serviceDetail Table

Field Name	Meaning
serviceID	Foreign key linking back to the mainRegistry table
parameterType	This value shows whether this parameter is an input or output parameter
parameterClass	The ontology concept of this given parameter

All these fields are quite obvious, and the only one that requires some explanation is the WSDLURL field. Note that even when a service provider semantically marks up the service description, there would still exist the WSDL document for the given service. It is a good idea to include a link to this WSDL document in the repository for reference. By the same token, you can include the URLs pointing to the process and grounding documents in case the description file is written using the OWL-S upper ontology. The point is, change the design as you wish and experiment with it to learn more about Semantic Web services.

As you can see, the mainRegistry table only saves the basic information about a given service (Table 1); the details about this service are stored in the serviceDetail table. Its structure is shown in Table 2.

III. SEMANTIC WEB

The basic idea of Semantic Web is that embed machine-readable, on behalf of certain types of knowledge mark in the Web message. So that the data on the Web is not only used to display, but also be understood by the machine so as to enhance the quality of the information services and explore a variety of new, intelligent information services. If the knowledge that reflect the link between data and application are embedded in a variety of different information sources in a user transparent manner, Web pages, database, procedures will be able to link up through the agent and each other collaborate. According to Berners-Lee's vision, the semantic network Constituted by seven levels is constituted of a layered architecture. As shown in Table 3.

Table 3: Semantic Web Architecture

	Layers	name	Description
Low ↕ High	Layer 1	Unicode and URI	The Semantic Web-based: Unicode Processing resources to encoding, URI (Uniform Resource Locator) negative Responsible for identification of resources
	Layer 2	XML+NS+XML Schema	Used to represent the data content and structure
	Layer 3	RDF+RDF Schema	Used to describe resources on the Web and types
	Layer 4	Ontology Vocabulary	Describe the various types of resources and the relationship between resources
	Layer 5	Logic	In the following four layers operate on the basis of logical reasoning
	Layer 6	Proof	According to logic, to verify statements in order to draw conclusions
	Layer 7	Trust	The establishment of a trust relationship between users

The first layer of URI and Unicode is the basis for the structure of the entire system. Unicode is responsible for processing resources encoding, Semantic Web is known as Web3.0, it is based on resource description framework RDF to integrate a variety of applications of XML-syntax, uniform resource identifier as naming mechanism. Semantic Web is just an extension of the current Web and is not a new Web. The research focus is how the information can only be changed from the form that a computer can read to the form that a computer can understand and deal with, that is with the semantics, so that the computer and people can work together. Web resources (such as Web pages, Web service) for the use of ontology annotation terms are an important prerequisite for goal to achieve the semantic Web. Ontology in Tim Berners-Lee proposed the Semantic Web-seven is in the fourth tier architecture, which aims to capture the knowledge in related fields, provides a common URI is responsible for resource identification, which allows precise retrieval of information possible.

The Second layer of XML + NS (Namespace) + XML Schema, is responsible for representing the content and structure of data from the linguistic to separate the performance format, the data structure and content of the network information form through the use of a standard format language.

The third layer of RDF + RDF Schema, which provides a semantic model used to describe the information on the Web and type.

The fourth layer of ontology vocabulary layer is responsible for the definition of shared knowledge and describes the semantic relationships between the various kinds of information to reveal the semantic between information itself and information.

The fifth layer of logic layer is responsible for providing axioms and inference principles to provide the basis for intelligent services.

The sixth layer of Proof and the seventh layer of trust are responsible for providing authentication and trust mechanisms. Digital signatures and encryption technology used to detect changes in the document situation is a mean to enhance Web security.

This is a hierarchical structure of the enhanced functional. XML, RDF (S) and the Ontology are its core in the Semantic Web architecture. The formation of the Semantic Web's technical support system mark with the three core technology. They support semantic description for network information and knowledge, to play a central role in achieving the semantic-level knowledge sharing and knowledge reuse.

IV. A MATH MAKING ALGORITHM

In this section, we will discuss a matchmaking algorithm in detail, which we will implement in the next section. A matchmaking algorithm is normally proposed in the following context:

• A domain-specific ontology is created to express the semantic capabilities of services, and these services are described with their inputs and outputs.

• For service discovery, clients and providers should use the same ontology to describe requests and services.

The first assumption is not new to us; we know that you have to markup your service description using some ontology. The second assumption is important to remember: if clients and providers use different ontologies, they

are then talking in different languages; there is no shared understanding of the context, and it is simply not possible for the matchmaking algorithm to work. Before we get into the details of the matching algorithm, let us first introduce some notations to simplify the presentation of the proposed algorithm. These notations are summarized in Table 4. Further, let C be any class or property, C = C(e), such that $\forall e \in (IR \cup IP \cup OR \cup OP)$, $C \in \Omega$. In other words, every input and output concept in the description files can be mapped to a single class or concept, namely, C, defined in the domain-specific ontology. Given all these definitions, we further need to define some useful mapping functions for the input set; these functions are listed in
Table 5.

Clearly, finput-exact represents an "exact" matching case. In other words, this exact matching can be expressed as follows:

- The number of input parameters required by the service is the same as that offered by the service requester.
- For each input class or concept in the requester's input parameter set, there is an equivalent input class or concept in the provider's input parameter set.

TABLE 4

Notations for Our Simple Matchmaking Algorithm

Variable Name	Meaning
I_R	The set of `input` concepts or classes provided by the service requester
O_R	The set of `output` concepts or classes provided by the service requester
I_P	The set of `input` concepts or classes in the service description file offered by the service provider
O_P	The set of `output` concepts or classes in the service description file offered by the service provider
Ω	The set of all the concepts or classes that are defined in the domain-specific ontology. Therefore, it is true that $I_R \subseteq \Omega$, $O_R \subseteq \Omega$, $I_P \subseteq \Omega$, and $O_P \subseteq \Omega$

TABLE 5

`Input` Mapping Functions Defined for Our Simple Matchmaking Algorithm

Function Name	Meaning
$f_{input\text{-}exact}$	A 1-1 mapping from $I_R \rightarrow I_P$: $\forall e_{iR} \in I_R$, $\exists e_{iP} \in I_P$, such that $C(e_{iR}) \equiv C(e_{iP})$. "$\equiv$" means the left-hand-side (LHS) concept is equivalent to the right-hand-side (RHS) concept; clearly, this implies that $\lceil I_R \rceil = \lceil I_P \rceil$, i.e., the number of input concepts given by the service requester is equal to that required by the Web service provided by the service publisher
$f_{input\text{-}L1}$	A 1-1 mapping from $I_R \rightarrow I_P$: $\forall e_{iR} \in I_R$, $\exists e_{iP} \in I_P$, such that $C(e_{iR}) < C(e_{iP})$. "<" means the LHS concept is a subconcept or subclass of the RHS concept; clearly, this implies that $\lceil I_R \rceil = \lceil I_P \rceil$
$f_{input\text{-}L2}$	A 1-1 mapping from $I_R \rightarrow I_P$: $\forall e_{iR} \in I_R$, $\exists e_{iP} \in I_P$, such that $C(e_{iR}) > C(e_{iP})$; ">" means the LHS concept is a superconcept or superclass of the RHS concept; clearly, this implies that $\lceil I_R \rceil = \lceil I_P \rceil$

$f_{input\text{-}L1}$ can be read and understood in a similar way. It is, however, not as desirable as finput-exact, and we thus call it a level-1 matching. More precisely, there exists some input parameter in the requester's parameter set that is a subconcept or subclass of the required class or concept. For instance, a service could be expecting Digital as the input parameter, but the service requester offers an SLR class, a subclass of Digital, as input. This is clearly not an exact match, but it is acceptable. To see this, look at the class concepts in the object-oriented design world: a subclass is simply a more specialized version of the parent class; therefore, an instance of the subclass should already contain all the data members that might be needed by an instance of the parent class. In other words, you can make an instance of the parent class from an instance of the subclass; so it is perfectly alright to accept an instance of the subclass where an instance of the parent class is needed.

Now it is easier to understand $f_{input\text{-}L2}$, the level-2 matching. It is just the opposite of level-1 matching: the required

input is a more specialized version than what the service requester can provide. For instance, a service could be expecting SLR as the input parameter, but the service requester offers a Digital class, a parent class of SLR, as input. This might still be a candidate because the service requester might have already provided enough information for the service to run. On the other hand, it is also possible that the service needs some specific information that an instance of Digital cannot provide.

If none of the foregoing three mapping functions hold when comparing the input sets, we can safely conclude that there is no match at all.

Similarly, three mapping functions can be analogously defined for the output concept set, as shown in Table 6.

TABLE 6

Output Mapping Functions Defined for Our Simple Matchmaking Algorithm

Function Name	Meaning
$f_{output\text{-}exact}$	A 1-1 mapping from $O_R \rightarrow O_P$: $\forall e_{OR} \in O_R$, $\exists e_{OP} \in O_P$, such that $C(e_{OR}) \equiv C(e_{OP})$. "$\equiv$" means the left-hand-side (LHS) concept is equivalent to the right-hand-side (RHS) concept clearly, this implies that $\lvert O_R \rvert = \lvert O_P \rvert$, i.e., the number of output concepts given by the service requester is equal to the number of input concepts required by the Web service provided by the service publisher
$f_{output\text{-}L1}$	A 1-1 mapping from $O_R \rightarrow O_P$: $\forall e_{OR} \in O_R$, $\exists e_{OP} \in O_P$, such that $C(e_{OR}) > C(e_{OP})$. ">" means the LHS concept is a superconcept or superclass of the RHS concept clearly, this implies that $\lvert O_R \rvert = \lvert O_P \rvert$
$f_{output\text{-}L2}$	A 1-1 mapping from $O_R \rightarrow O_P$: $\forall e_{OR} \in O_R$, $\exists e_{OP} \in O_P$, such that $C(e_{OR}) < C(e_{OP})$; "<" means the LHS concept is a subconcept or subclass of the RHS concept clearly, this implies that $\lvert O_R \rvert = \lvert O_P \rvert$

A Simple Matchmaking Algorithm

```
Input: 1. Web service request description (.owl)
       2. current service description from the repository
Output: a string value from the set {"exact","level-1",
"level-2","failed"} Method: build I_R,O_R using Web service
request description file; build I_P,O_P using the current
SWS repository; if |I_R| ≠ |I_P| or |O_R| ≠ |O_P| return
"failed"; else if ( ∃f_input-exact and ∃f_output-exact ) return
"exact"; else if ( ∃f_input-exact and ∃f_output-L1 ) return
"level-1"; else if ( ∃f_output-exact and ∃f_input-L1 ) return
"level-1"; else if ( ∃f_input-L1 and ∃f_output-L1 )   return
"level-1"; else if ( ∃f_input-exact and ∃f_output-L2 ) return
"level-2"; else if ( ∃f_output-exact and ∃f_input-L2 ) return
"level-2"; else if ( ∃f_input-L1 and ∃f_output-L2 )   return
"level-2"; else if ( ∃f_input-L2 and ∃f_output-L1 )   return
"level-2"; else if ( ∃f_input-L2 and ∃f_output-L2 )   return
"level-2"; else return "failed";
```

Fig. 1: A Simple Matchmaking Algorithm

Again, this algorithm is easy to understand; in the next section, you will see an implementation of this algorithm. The basic idea is to first check the number of inputs and outputs; if the number of inputs (or outputs) of the required service is different from the number of inputs (or outputs) of the current candidate, the match immediately fails. If the numbers match, the algorithm goes on to check the mapping functions to determine the degree of matching. For instance, if exact matching functions can be found for both the inputs and outputs, then the current candidate service exactly matches the requirement. But if exact matching functions can be found for inputs (or outputs), but only level-1 matching functions can be found for outputs (or inputs), then the matchmaking engine will declare a level-1 matching, and so on.

Note that this simple matching algorithm considers mainly the matching between inputs and outputs. Other factors can also be taken into account. For example, the service category can also be considered a matching criterion, and it is not hard to extend this to the aforementioned algorithm. Also, you can take preconditions and effects of the Web service into consideration. We will not cover all these possibilities in this chapter; this algorithm serves as a starting point for your journey of creativity. As we have discussed earlier, level-1 and level-2 matching results were proposed to differentiate the parent–child relationship of the concept. In the literature, this relationship is sometimes called subsumption relationship. It is also obvious that the direction of this subsumption relation is important: in the case where input A in the request profile subsumes input

B in the candidate service profile, the advertised service may require some specific details for proper execution that input A cannot provide. In a situation like this, our matching algorithm still includes this service as a potential candidate and lets the service requester decide whether the service is usable or not. This degree of matching is called a level-2 matching; therefore, a level-2 matching may not be as appropriate as a level-1 matching.

Up to this point, we have discussed all the major components in the search engine; it is now time to do some implementation work.

V. IMPLEMENTATION OF THE SEARCHING FUNCTIONALITIES

We have already accomplished a lot by building the crawler and the repository that holds the descriptions of the Semantic Web services. However, to make this search engine work, some more work needs to be done.

First, we have to build an architecture to hold the entire system. Clearly, the crawler and the repository have to live on the server, and the component that implements the matchmaking algorithm has to be on the server too. On the other hand, there has to be a client that accepts the user request, somehow invokes the search engine, and also presents the search result to the user. Currently, this overall architecture is unclear. Let us discuss this architecture first and then get into the details of the implementation of the

searching functionalities.

5.1 Suggested Architecture for Testing

Before we set up the whole client-server architecture, we have to keep the following points in mind:

- The crawler lives on the server; when invoked by the controller component, it will crawl to the Web to collect the Semantic Web service descriptions and store them in the repository.
- The repository lives on the server; it is supported by a database system (in our case, we are using Microsoft Access as our back-end database system).
- The actual searching component, or the matchmaking engine, resides on the server and has access to the repository.
- The server itself will listen to the incoming calls from the client; once it receives an incoming call, it will invoke the

matchmaking engine and send the response (search results) back to the client.

- The client is a Web browser-based client; in other words, the front end of the search engine has the look and feel of an ordinary search engine. It is therefore a very lean client: it is only responsible for accepting user search requests and posting the requests back to the server; the returning results are rendered by the Web browser to the users.
- Given that the client is Web browser based, the incoming calls to the server are HTTP GET or HTTP POST calls, and responses sent back by the server are in the form of a HTTP response stream.
- The components living on the server are implemented using Java.

The server has to be a Web server to satisfy all these criteria. More specifically, this Web server should have the ability to support the HTTP protocol; once it receives an incoming search request from the client, it should invoke some servlets to conduct the search and produce a HTML page that contains the results. Clearly, the matchmaking engine has to be implemented using servlets technology.

Let us now take a closer look at how we can implement such a prototyping system. I assume you will test our

Semantic Web service search engine using your development PC either at your home or your office. In other words, there is no dedicated server box created just for testing and prototyping purposes. Therefore, the most practical architecture is to use your development machine as both the server and the client at the same time.

When your development machine is used as the server, localhost will be its name. The following steps will set up your localhost as a Web server:

1. Download Sun Java System Application Server Platform and install it.
2. Start the server; this will make your localhost a Web server capable of handling incoming HTTP GET or HTTP POST calls; now your localhost also supports servlets.
3. Once the server starts, note the port number it is listening to; you need to use this port number to call the server. You can change it if you like.

These steps are listed here to give you an idea of how to set up the architecture needed by our search engine. I will not discuss the details, such as where to download the Sun Java System Application Server, how to install it, or how to configure it (change the port number it uses). You can always follow the instructions on the Sun Web site to perform these tasks.

Now we have finished setting up our server. The next step is to set up the client on the same machine, to complete the testing architecture. This task is fairly simple: all you need to do is to construct a HTML page that takes the user's search request.

VI. CONCLUSION

We accomplished this goal by a detailed review of the current standards for Web services, including WSDL, SOAP, and UDDI. More speci fically, we concentrated on the internal structure of the UDDI registry, especially the service discovery mechanism provided by UDDI. This led to the conclusion that automatic service discovery is almost impossible to accomplish if we solely depend on UDDI registries and current Web service standards. To facilitate automatic discovery, composition, and monitoring of Web services, we need to add semantics to current standards. Before we can add semantics to current Web service standards, we have to design a language that we can use to formally express the semantics of a given Web service. OWL-S is the current standard language for expressing Web service semantics. WSDL is also used to describe Web services, understanding the relationship between WSDL and OWL-S is important for Semantic Web developers. The main idea is to build a Semantic Web service search engine that manages its own repository that does not depend on public UDDI registries. I presented a detailed design of such a search engine and also showed the implementation of its key components using Java programming together with Jena APIs. By developing a working Semantic Web services search engine prototype. The programming skills presented here were the most frequently used ones in real-world applications, and the code presented can certainly be reused in your future development work.

REFERENCES

[1] Introduction to the Semantic Web and Semantic Web Services by Liyang Yu,2007
[2] Wen-Wei Chen, "Data Warehouse and Data Mining Tutorial,"[M], Beijing: Tsinghua University Press, 2008, 4
[3] Zhong Xue Ling, "Semantic Web in the core layer of technical analysis,"[M], South China Financial Computer Applications Technology, 2007,10
[4] Lu Jian-Jiang, "Semantic Web principles and technology," [M], Beijing: Science Press, 2007, 3
[5] Zhang Hui, ed, "Ontology-based Semantic Web Mining Technology."[D], computer development and pplications, 2009.