Preliminary Report of Public Experiment of Semantic Service Matchmaker with UDDI Business Registry

Takahiro Kawamura[1], Jacques-Albert De Blasio[1], Tetsuo Hasegawa[1], Massimo Paolucci[2], and Katia Sycara[2]

[1] Research and Development Center, Toshiba Corp
[2] The Robotics Institute, Carnegie Mellon University

Abstract. The public experiment of the semantic service search with the public UDDI registry is shown in this paper. The UDDI is a standard registry for Web Services, but if we consider it as a search engine, the functionality is restrictive, that is, based on keyword retrieval only. Therefore, the Matchmaker was developed to enhance the UDDI search functionality by using semantics such as ontology and constraints. However, for Web Services as e-Business platform, compliance with the standard specification such as SOAP, WSDL, UDDI is the key. Thus, our goal of this experiment is to seamlessly combine semantic search with those standards, and investigate the feasibility of using semantics with Web Services. This paper firstly shows the overall architecture where the Matchmaker can be located between UDDI and service developers/users. Then, the Matchmaker and our semantic service description which can be complementary used with the standard WSDL and UDDI Data Structure are shown. Also, we illustrate some tools which generate the service description and support the use of the Matchmaker. Finally, we qualitatively evaluate this experiment on the response from business users.

1 Introduction

Web Services are considered as the core technology of e-Business platforms. The spreading of Web services in the Intranets and in the near future in the whole Internet reveals the needs of sophisticated discovery mechanisms. In the space of discovery, UDDI is emerging as the de-facto standard registry for Web services and it is proposed as the main tool for Web service discovery. However, the only discovery mechanism provided by UDDI is keyword search on the names and the features of businesses and services descriptions; unfortunately, keyword search fails to recognize the similarities and differences between the capabilities provided by Web services. Ultimately, UDDI is useful only to find information about known Web services, but it completely fail as a general Web services discovery mechanism.

To address this problem, we developed Semantic Services Matchmaker, a search engine for Web services, that enhances the discovery facilities of UDDI

M.E. Orlowska et al. (Eds.): ICSOC 2003, LNCS 2910, pp. 208–224, 2003.

to make use of semantic information. Furthermore, we initiated an experiment on a publicly available UDDI maintained by NTT-Communications, one of four official UDDI operators to evaluated the scalability and viability of our approach on a large scale[1].

The Semantic Services Matchmaker is based on the LARKS[1] algorithm, but it also borrows some ideas from the DAML-S matching algorithm[2]. Specifically,it adopts the LARKS filtering approach which uses sophisticated information retrieval mechanisms to locate Web services advertisements in UDDI even when no semantic information has been provided. To this extent, we can show the contribution and the limits of information retrieval techniques to Web services discovery, and the contribution of ontological information to increase the precision of matching.

In the rest of the paper we will discuss our approach and experiment. In section 2 we first introduce the overall architecture of this experiment; in section 3 and 4 we will introduce the Semantic Service Matchmaker and its semantic service description called Web Services Semantic Profile (WSSP). In section 5 we describe supporting tools that provide an interface to UDDI that allows semantic markup and matching. Then, in section 6, we will present an initial evaluation on the response from business users we have collected during its design and deployment phase.

2 Architecture of Public UDDI Registry and Matchmaker

The architecture of the integrated UDDI Registry and Matchmaker is shown in Fig. 1. In the architecture, the Matchmaker is inserted between the users of the matchmaker, namely users that request services and Web Services developers that advertise them, and the UDDI registry. The Matchmaker API is equivalent to the UDDI API to facilitate the users seamless connection; furthermore, the format of the results returned adopts the same format of the results returned by UDDI. The result is that the Matchmaker can be added to any UDDI registry leaving to users and developers the choice of the most preferable Web Services registry in accordance with the cost and interoperability with their already-existing systems. Fig. 2 shows the internal architecture of the Matchmaker, which will be discussed in details in section 3.

The second feature of the architecture is that it strives to be compliant with the current standard technology of Web Services such as SOAP and WSDL, since the use of any proprietary technology would reduce the advantage of using Web Services as the common e-Business platform.

[1] The Semantic Services Matchmaker has been implemented in a collaboration between Carnegie Mellon University and Toshiba Corp. The public experiment is a collaboration between Toshiba Corp. and NTT-Communications. The experiment site is at www.agent-net.com.

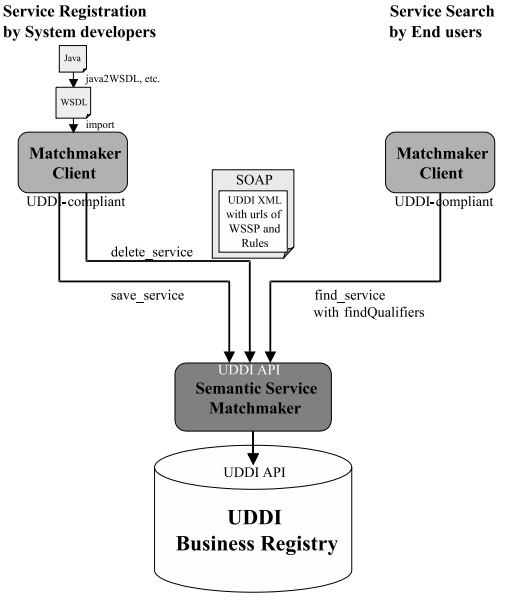


Fig. 1. Network architecture of UDDI and Matchmaker

2.1 Usage Scenario

In the rest of this section we will describe how the Matchmaker is used, and specifically we will describe service registration and search scenarios. We hope to demonstrate the contribution of the Matchmaker to the UDDI search facilities.

Registration

1. During service registration the client tools associated with the Matchmaker Client in Fig. 1, are used to generate a semantic service description from a WSDL description which has been automatically generated from the java source code. The client tools, which will be described in details in section 5, support annotation of the ontology classes in RDFS[8], DAML+OIL[9], or OWL[10] to inputs and outputs' parameters, and also the definition of rules as inputs and outputs' constraints. Of course the client may decide not to send any semantic description to the Matchmaker at all; in the latter case, the client registers as a standard UDDI client.
2. Upon receiving the registration, the Matchmaker extracts the semantic annotation. If the annotation is found, the Matchmaker stores it with all the ontologies it refers to. Finally, the Matchmaker registers the service with the UDDI registry.

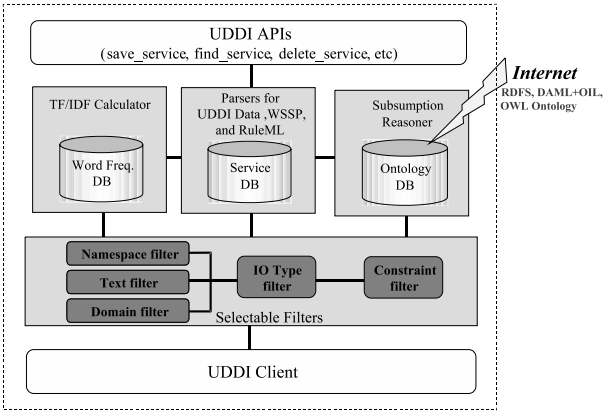


Fig. 2. Semantic Service Matchmaker

Search

1. During the service search, the client sends a search request to the Matchmaker. The search request consists of a search in UDDI augmented with a semantic annotation of the desirable service interface, consisting of all the desirable inputs' ontology classes and outputs' ontology classes. As in the case of the service registration, users can specify the inputs and outputs' constraints. Furthermore, the client may also decide to restrict his search to the requirements of a normal UDDI keyword search tool.
2. The Matchmaker checks whether or not the search request includes semantic designation like ontology classes and rules. If so, the matching engine searches for a service that is similar enough to the service requested in the registered services database. After making the matching results, the Matchmaker retrieves the detailed information of those results from the UDDI registry, then get back them to the client. Note that several search options for the Matchmaker can also be specified via UDDI APIs.

3 Semantic Service Matchmaker

Above we showed how users and programs can exploit the Matchmaker and the UDDI API to find the services that provide the capabilities that they expect. In this section, we provide the details of how the matching of capabilities is performed.

Ideally, when the requester looks for a service, the matchmaker will retrieve a service that matches exactly the service that the requester expects. In practice, it is very unlikely that such a service is available, instead the matchmaker will retrieve a service whose capabilities are *similar* to the capabilities expected by the requester. One of the challenges of matchmaking is to locate services that the requester could despite the differences from the request. Furthermore, the

matchmaker should be able to characterize the distance between the request and the matches found, so that the requester can make an informed decision on which services to invoke.

To address this problem, the matchmaker identifies the three levels of matching where *exact* corresponds to an exact match between the request and the services provided. Furthermore, we allow the requester to specify how closely should the request and the matches provided be. Ultimately, the matching process is the result of the interaction of the services available, and the requirements of the requester.

Exact match is the highest degree of matching, it results when the two descriptions are equivalent.

Plug-in match results when the service provided is more general than the service requested, but in practice it can be used in place of the ideal system that the requester would like to use. To this extent the result can be "plugged in" place of the correct match. A simple example of a plug-in match is the the match between a requested service that sell books and a service that sells printed materials. Since books are printed materials chances are that the latter service can be used instead.

Relaxed match. The relaxed match has a weakest semantic interpretation: it is used to indicate the degree of similarity between the advertisement and the request.

The second aspect of our matchmaker is to provide a set of filters that help with the matching, and we allow users to decide which filters they would like to adopt at any given time. Specifically, the matching process is organized as a series of five filters. All filters are independent from each other, and each of them narrows the set of matching candidates with respect to a given filter criterion. The first three filters are meant for the relaxed match, and the last two are meant for the plug-in match (see Fig. 3). Users may select any combination of these filters at the search time, considering the trade-off between accuracy and speed of the matching process. We briefly illustrate each filter as follows. Further details are provided in Sycara et al.[1].

3.1 Namespace Filter

Pre-checking process which determines whether or not the requested service and the registered ones have at least one shared namespace (a url of an ontology file). The intersection of namespaces can be considered shared knowledge between the request and the advertisement. Therefore, only the registered services which have at least one shared namespace go into the next filter. Namespaces of default like rdf, rdfs, xsd, etc. are not considered the intersection. Of course, there is a case that the relation between two nodes of different ontology files does exist, although the distance would be relatively long. This filter and the next two filters are meant for the reduction of the computation time of the last two filters.

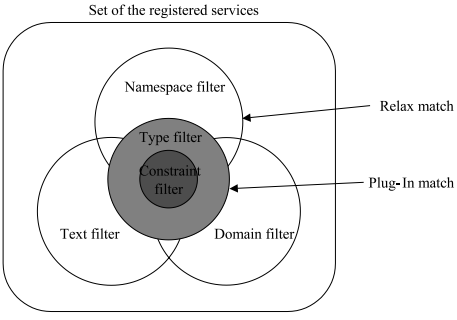


Fig. 3. Criterion of each filter

3.2 Text Filter

Pre-checking process for human-readable service explanation parts such as comment and text descriptions. It utilizes the well-known IR (Information Retrieval) technique called TF/IDF (Term Frequency Inverse Document Frequency) method. This filter help to minimize the risk to miss the services which have any relation to the requested one.

3.3 Domain Filter

Pre-checking process to check whether or not each registered service and the requested one belong to an ontology domain. The ontology domain here means a subtree in a ontology tree. To determine the subtree which two services belong to, we first we extract ontology nodes related with the service category or outputs as concepts of the services, then we select a common ancestor in the ontology tree. If the advertisement and the request are in a certain size of the subtree, the registered one passes the filter. Note that if we can find relationship such as subClassOf, sameAs between different ontology trees, we merge the trees and trace up them seamlessly.

3.4 I/O Type Filter

The Type Filter checks to see if the definitions of the input and output parameters match. In the semantic service description shown in the next section, parameter types of inputs and outputs are defined as ontology classes. A set of subtype inferencing rules mainly based on structural algorithm are used to determine in this filter.

Such a match is determined if: (see Fig. 4)

- the types and number of input/output parameters exactly matches, OR
- outputs of the registered service can be subsumed by outputs of the requested service, and the number of outputs of the registered service is greater than the number of outputs of the requested service, AND/OR

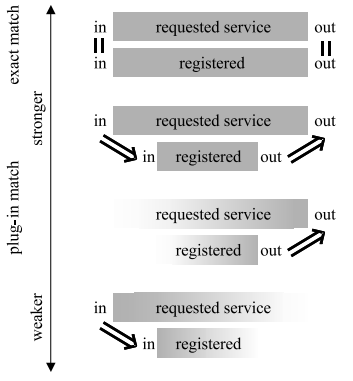


Fig. 4. Levels of plug-in match

– inputs of the requested service can be subsumed by inputs of the registered service, and the number of inputs of the requested service is greater than the number of inputs of the registered service.

If there is a mismatch in the number of parameters, then the filter attempts to pair up parameters in the registered one with those in the request, by seeking the registered one's parameters that are sub-types of the requested one's parameters.

Further, the request may not have a model of what inputs may be required, and may need to obtain this information from the returned service. To support this, inputs and inputs' constraints in the next section also match when those of the request are empty.

3.5 Constraint Filter

The responsibility of this filter is to verify whether, the subsumption relationship for each of the constraints are logically valid. The constraints filter compares the constraints to determine if the registered service is less constrained than the request. The Matchmaker computes the logical implication among constraints by using polynomial subsumption checking for Horn clauses. Matching is achieved by performing conjunctive pair-wise comparisons for the properties. In detail, the logical implication among constraints is computed using polynomial θ-subsumption checking for Horn clauses.

The constraints for inputs and outputs are defined for the request (R_I and R_O) and for each registered service (A_I and A_O). The constraints for inputs R_I is compared with A_I, and a match is determined if A_I subsumes R_I, i.e.

$$match(R_I, A_I) \Leftarrow (\forall j, \exists i : (i \in R_I) \wedge (j \in A_I) \wedge subs(j, i)) \vee R_I = \emptyset$$

where $subs(j, i)$ is true when j subsumes i. The constraints for outputs match when all the elements in R_O subsumes elements in A_O, i.e.

$$match(R_O, A_O) \Leftarrow \forall i, \exists j : (i \in R_O) \wedge (j \in A_O) \wedge subs(i, j)$$

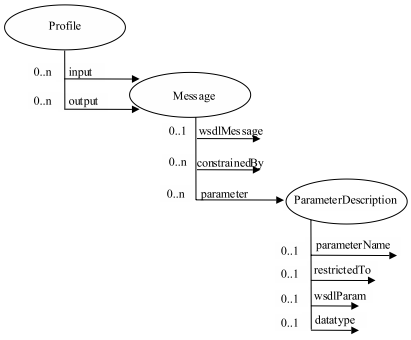


Fig. 5. Structure of WSSP

4 Semantic Service Description

In the previous section we described the matching process, and specifically we described how ontologies are used by the constraint filter. In this section, we describe the Web Service Semantic Profile (WSSP), a way to encode semantic information in WSDL that is inspired by the DAML-S Service Profile[11].

WSDL specifies the programming interface of Web service to be used at invocation time. To this account it specifies the name of the function to invoke and the type of data that the Web service expects as input or it generates as output. Data types describe how the information is formatted, but because there are arbitrarily many ways of encoding the same information in data types, they fail to express the semantics of the information that they encode. Unfortunately, semantic information is exactly what is needed to express the capabilities of Web services. The aim of the WSSP is to enrich WSDL with the semantics it needs to represent capabilities of web services.

Fig. 5 shows the structure of a WSSP, and its relation with WSDL. As in a WSDL file, we define each message and each parameter of those messages. *Parameter descriptions* store descriptions of parameters that are used to define the semantics of the parameter in the *restrictedTo* element as well as the data types of the information in the *datatype* element, as shown in the example below.

Moreover, we offer the possibility to add constraints to the inputs and outputs of the web service. Doing so we give the possibility to represent the web service more accurately, and we allow the search engine to perform a more accurate search. Constraints are added to the WSDL specification through the element *constrainedBy* which specifies the URI of a fact or rule written in an RDF-RuleML file [7].

The relation between the facts and rules and the WSSP are simple: each parameter of an input or output of the web service corresponds to a variable which can be used in facts and rules. It means that if a web service has “n” inputs parameters and “m” output parameters, the number of available variables will be “n+m”. Thus we can write constraints using the inputs and outputs of

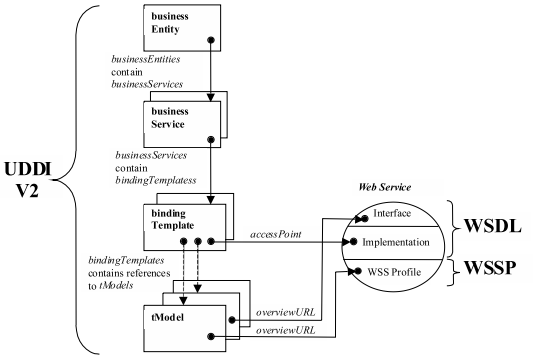


Fig. 6. Relationship between UDDI, WSDL and WSSP

the web service. Note that facts and rules are not bound to any input/output of the web service, they just uses them.

An example of an input message of a WSSP is given below. It presents examples of use of all the elements described above, and it also shows the relation to the original WSDL file through xPointer links [6].

```
<profile:input>
  <profile:message rdf:ID="1st_INPUTMessage">

    <profile:parameter>
      <profile:ParameterDescription rdf:ID="1st_INPUTMessage_Param_1">
        <profile:parameterName>Param_1</profile:parameterName>
        <profile:restrictedTo rdf:resource="http://ont.com/Onto.owl#Item1"/>
        <profile:wsdlParam>xPointer to wsdl parameter</profile:wsdlParam>
        <profile:datatype>XMLSchema.xsd#String</profile:datatype>
      </profile:ParameterDescription>
    </profile:parameter>

    <profile:constrainedBy rdf:resource="http://rule.com/rules.rdf#1"/>
    <profile:constrainedBy rdf:resource="http://rule.com/rules.rdf#2"/>

    <profile:wsdlMessage>xPointer to wsdl message</profile:wsdlMessage>

  </profile:message>
</profile:input>
```

In Fig. 6, we describe the relation between UDDI, WSDL and WSSP. A Web Service is completely described by three entities, the WSDL interface and binding (implementation) and the WSSP. This architecture provides a clear separation between how the Web Service works and what it does. Moreover, as we mention later, there is no redundancy between the data recorded in the UDDI description of the Web Service and a WSSP.

5 Supporting Tools

In this section, we introduce our client tools for the registration and search of services through an example. These tools allow human users to register Web

services with the Matchmaker and UDDI, as well as search for Web services. The example we use concerns a web service that can tell a requester in how many days a given manufacturer can deliver a certain type of material (called parts in our example). This web service has first been described via a WSDL file which may have been generated automatically from Java source code. In this example, the user will first import the WSDL interface in *Matchmaker Client* and then add semantic annotations to input and output parameters using *Ontology Viewer*. Furthermore, the user adds facts and rules in order to add input constraints by using *Rule Editor*

The following is the part of the WSDL interface containing the description of the input and output messages of the web service.

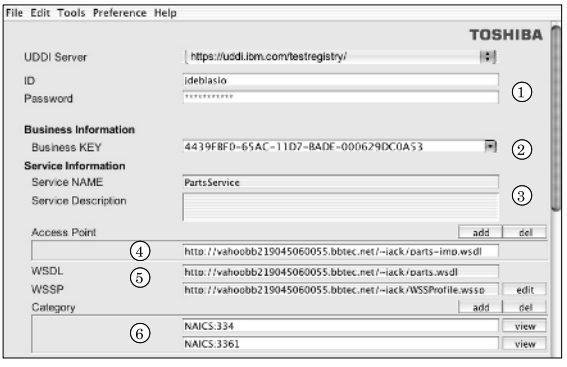
```
<message name="PartsRequest">
  <part name="part" type="xsd:string"/>
  <part name="numberParts" type="xsd:int"/>
  <part name="manufacturer" type="xsd:string"/>
  <part name="country" type="xsd:string"/>
</message>

<message name="DeliveryResponse">
  <part name="deliveryDays" type="xsd:int"/>
</message>

<portType name="PartsService">
  <operation name="getQuote">
    <input message="sq:PartsRequest"/>
    <output message="sq:DeliveryResponse"/>
  </operation>
</portType>
```

5.1 Matchmaker Client

The following steps illustrates a typical use of the Matchmaker Client. Note that because the Matchmaker has the same API as the UDDI registry, the Matchmaker Client can be used as a UDDI client tool, and also other UDDI client tools can be used to register and search to the Matchmaker, but in this case they require the creation of WSSP with the appropriate semantic information.

Creation of a Business Service. The tools for the creation of the *Business Service* are shown in Fig. 7,8. To create a Business Service the user can import the WSDL file in the Matchmaker Client, completing automatically some of the fields. The user can add the ontology annotations using the Ontology Viewer, which parses ontology files written by RDFS, DAML+OIL, or OWL, then show them as graphical trees. The user can specify any ontology class to be annotated to each parameter by clicking a node in the trees (see Fig. 9). Finally, the user can add facts and rules using the Rule Editor, described in details below.

Registration of the Business Service. After creating a Business Service, the user can register it with the Matchmaker and UDDI as a consequence. Fig. 10 shows that the WSSP is created automatically as part of the registration. The WSSP is then uploaded on the Matchmaker with all the ontology files and the definitions of the rules.



1. Information about the UDDI Server and the user's login and password
2. Business Key corresponding to the business entity for which the user wants to register this business service
3. **Auto-completed** fields after having read the WSDL file
4. URL to implementations of the web service
5. URLs to the WSDL and WSSP files (the latter must be uploaded by the user once automatically created)
6. List of categories under which the business service can be classified

Fig. 7. Service registration - part1

An example of WSSP generated by the Matchmaker Client is shown below. There one of the inputs is a `part` that is described semantically by the concept `Parts.owl#Part` and syntactically specified as a `XSD:String`. Furthermore, the example shows a set of rules that provides additional properties that we want that part to hold. As pointed out above, xPointers are used to relate the WSSP to the WSDL. The precise definition has been omitted here to make the code more readable.

```
<profile:input>
  <profile:Message rdf:ID="PartsRequest">

    <profile:parameter>
      <profile:ParameterDescription rdf:ID="PartsRequest_part">
        <profile:parameterName>part</profile:parameterName>
        <profile:restrictedTo rdf:resource="http://example.com/onto/Parts.owl#Part"/>
        <profile:wsdlParam>xPointer to this parameter in the WSDL file</profile:wsdlParam>
        <profile:datatype>http://www.w3.org/2001/XMLSchema#string</profile:datatype>
      </profile:ParameterDescription>
    </profile:parameter>

    [other input parameters...]

    <profile:constrainedBy rdf:resource=http://bb.net/rules.rdf#ManufacturerOf(Part,Acme)/>
    <profile:constrainedBy rdf:resource=http://bb.net/rules.rdf#ManufacturerOf(Toshiba,Computers)/>
    <profile:constrainedBy rdf:resource=http://bb.net/rules.rdf#CountryManufacturer(Toshiba,JAPAN)/>
    <profile:constrainedBy rdf:resource=http://bb.net/rules.rdf#ManufacturerOf(Ford,Cars)/>
    <profile:constrainedBy rdf:resource=http://bb.net/rules.rdf#CountryManufacturer(Ford,USA/>
    <profile:wsdlMessage>xPointer to this message in the WSDL file</profile:wsdlMessage>

  </profile:Message>
</profile:input>
```

Search of the business service. Finally, Fig. 11 specifies the search tools that we defined. They allow to specify the number of inputs and output of the desired service, to choose ontologies for each parameter through the Ontology Viewer, and finally, to select facts and rules with the Rule Editor.

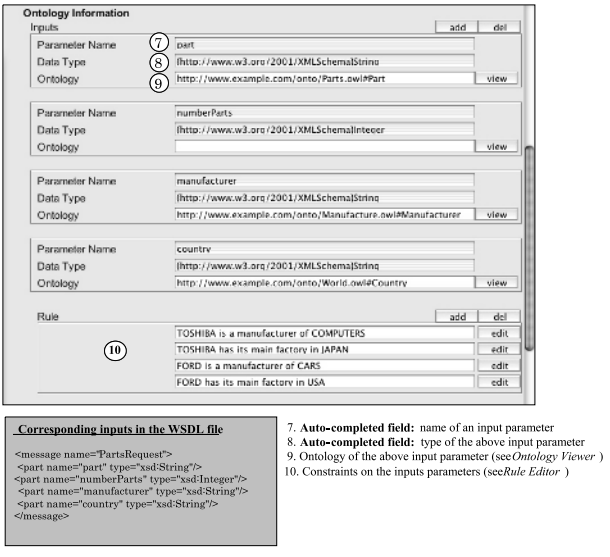


Fig. 8. Service registration - part2

5.2 Rule Editor

The writing of RuleML rules is a difficult and wordy process even for logic experts. The example of fact below, shows a very simple statement: `CountryManufacturer(Toshiba,JAPAN)` takes 16 lines of XML code to be expressed. While RuleML adds value and expressive power to WSSP, it is bound to be unusable unless we facilitate the process of creating rules. To address this problem, we developed a tool called the Rule Editor which allows anyone to easily write facts and rules, as if they were writing natural sentences.

The Rule Editor is invoked by the user by pressing the "edit" button when registering a service or searching for one. The invocation of the Rule Editor will result in the interface shown in Fig. 12 and the user will be able to easily create facts and rules. He will then choose one of the fact or rule and click on the "Send to WSMM" button. Thus doing, the id of fact or rule will appear in the "Rule" text field of the Matchmaker Client, and a RDF-RuleML file will be automatically generated.

The variables used in the facts and rules correspond to the input and output parameters of a WSDL file. Following is a part of the file generated by the Rule Editor (the part corresponds to the selected fact on the snapshot). Note the difference between how the fact is represented in the Rule Editor and how it is actually written in the RDF-RuleML file. The correspondence between the two is realized via an external file that tells the Rule Editor how to read predicates. For instance there would be, in this definition file, a line stating that the predicate "CountryManufacturer" is read "has its main factory in" and that it has two terms. This file would have been created by an expert. Eventually, creating facts

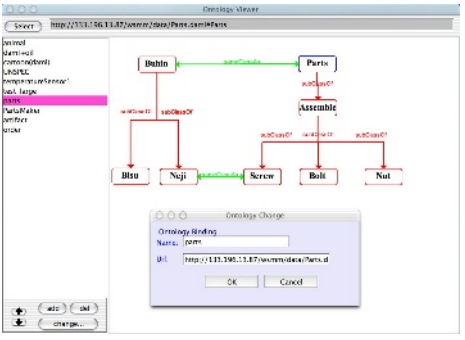


Fig. 9. Ontology Viewer

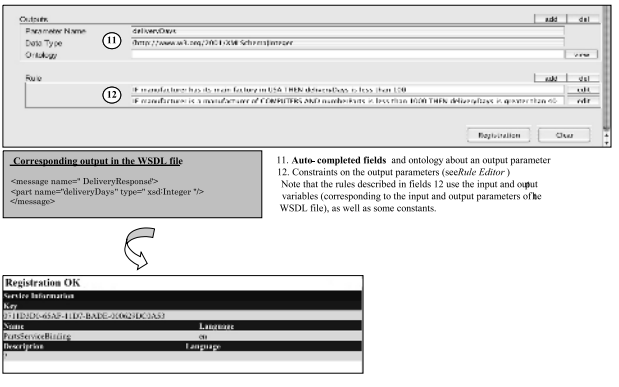


Fig. 10. Service registration - part3

and rules will become easy to any user, because writing those using our Rule Editor will be as natural as to write simple English sentences.

```
<ruleml:Fact ruleml:label="CountryManufacturer(Toshiba,JAPAN)">
  <ruleml:head>
    <ruleml:Atom ruleml:rel="http://example.com/preds/Predicates.owl#CountryManufacturer">
      <ruleml:args>
        <rdf:Seq>
          <rdf:li>
            <ruleml:Ind ruleml:name="http://example.com/onto/Manufacture.owl#TOSHIBA" />
          </rdf:li>
          <rdf:li>
            <ruleml:Ind ruleml:name="http://example.com/onto/World.owl#JAPAN" />
          </rdf:li>
        </rdf:Seq>
      </ruleml:args>
    </ruleml:Atom>
  </ruleml:head>
</ruleml:Fact>
```

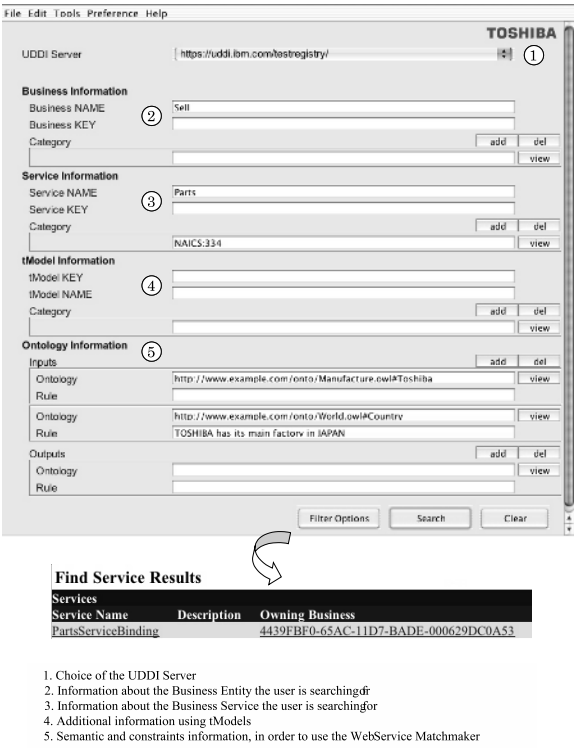


Fig. 11. Service Search

6 Evaluation

The Matchmaker service described in this paper, which provide a semantically-enhanced UDDI Business Registry has started its operations only recently, therefore we can report only a qualitative evaluation of our system design.

At the design and deployment phase of this experiment, we have collected lots of VoC (Voice of Customer) from system integrators and user companies of Web Services, by using DFACE (Define-Focus-Analyze-Create-Evaluate) methodology, a version of "Design for Six Sigma"[12] developed by Stanford University.

The responses of the users in terms of requirements for Web Services are as follows. The No.1 request from both of system integrators and users is the reduction of the cost of development and administration. The second requirement is interoperability with the current systems running or selling in their companies. The third one is a track record and security in general. In term of Web Services search engine itself, the most important thing is ease of use and search speed, rather than advanced functions.

Those results made us decide to obey the standard technology as much as possible. This means not only the improvement of the interoperability, but also

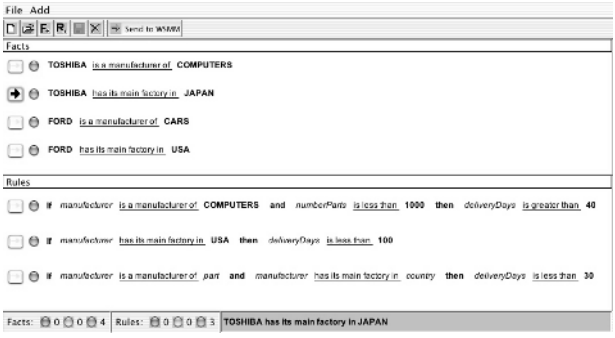


Fig. 12. Rule Editor

the reduction of the cost because it will give users and developers freedom of choice at the selection of the private UDDI registry product and the client tools. Besides, we expect this public experiment contributes to the track record. Further, we expect that client tools including the Rule Editor and the Ontology Viewer will lower the threshold of use of semantics.

Here we should note that from the point of the interoperability WSSP is a weakness of our system since it does not correspond to any existing Web service description language. However the combination of of Web Services and semantics is a kind of missing link at this time. Therefore we had to invent it as a glue. Although we adopted WSSP in this experiment, if any standard will allow the description the same contents as the WSSP in near future, we will be pleased to adopt it.

As future work, we would like to present the quantitative evaluation of this whole experiment in terms of usability, accuracy, speed, and so forth.

7 Related Work

As we mentioned above, the Matchmaker is based on the LARKS[1] and the DAML-S matchmaker[2]. Here we briefly describe other approaches of the matchmaking problem.

InfoSleuth[4] is a multi-agent system that supports information discovery and retrieval application where broker agents provide semantic brokering. Service provider agents advertise their service capabilities and constraints to the broker in LDL++, a logical deduction language developed by MCC, the InfoSleuth company. The requester agents query for the service providers to the broker in the same language. Then the broker uses constraint-based reasoning to find agents whose services match the constraints specified by the requester. In contrast, we prefer to close to Web Services standards such as WSDL and UDDI, then complement them with Semantic Web standards such as OWL and RuleML.

Horrocks et al.[5] have develped a framework for matchmaking based on Semantic Web technology like DAML-S[11]. Their matchmaker uses a DL (De-

scripion Logics) reasoner to match service arvertisements and requests based on the semantics of ontology-based service description. Then, the performance evaluation shows that DL reasoning technology could cope with large scale e-commerce application. We can see similarities between their matchmaker and our work, since their work is based on the LARKS to some extent. Although their DL reasoner is highly organized, we are using not only the DL technology but also the information retrieval technique for no semantic information. Beside, as with the above, our work differs from their system in the aspects of adherence to Web Services standards.

In terms of the services description language, WSSP has been greatly inspired by the work done for the DAML-S Profile[11], although our work reflects a different point of view which is one of our main goals, closer to the industry needs.

DAML-S is meant to describe everything a web service can do by combining a Service Profile, a Service Model and a Service Grounding. This approach is very powerful because all the details about a web service, from the most general down to the smallest, can be fully described. Yet, because of the relations between the different modules of DAML-S, it is difficult to use only the Service Profile without implicitly adopting the Process Model and the Grounding. On the other side, the industry is fragmented with different standards emerging, so any commitment to one technology or the other may prove very dangerous at this point. In this work we adopted the basic ideas of DAML-S, and we imported them directly in WSDL which is the minimum common denominator of all the Web services technology.

One of the goals of this search engine was to work as a semantic layer on top of UDDI. The results of a search would be a list of one or more UDDI Business Service. The solution proposed for DAML-S to work along with UDDI implied the mapping of the DAML-S Profile to a UDDI Data Structure[3]. Although this solution is elegant, we would have redundancy if we map the data of the DAML-S Profile and WSDL to the UDDI Data Structure. Thus, we decided to make our profiles "lighter" and avoid repeating descriptions between our profiles and WSDL.

8 Conclusion and Future Work

In this paper, we provides an initial report on our public experiment implementing the Semantic Service Matchmaker to expand the functionalities of the NTT-Communications public available UDDI registry. Here we described the architecture and the functionalities of the Matchmaker, and specifically the matching process. Furthermore, we provide a description of the WSSP, an extension of WSDL to describe capabilities of Web services, and the supporting tools implemented guided by business users' voice.

This is the first step toward the experiment, and that we did it on the bases on the interaction with the users and their comments. As the future works, we do need to make the quantitative evaluation after getting some amount of users' records, then tune the performance and functionality of the Matchmaker, and

usability of the supporting tools. Further, although this is not our own issue, we should tackle the problem of ontology definition and management to facilitate the use of ontology.

References

1. K. Sycara, S. Widoff, M. Klusch, J. Lu, "LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace". In Autonomous Agents and Multi-Agent Systems, Vol.5, pp.173–203, 2002.
2. M. Paolucci, T. Kawamura, T. R. Payne, K. Sycara, "Semantic Matching of Web Services Capabilities", Proceedings of First International Semantic Web Conference (ISWC 2002), IEEE, pp. 333–347, 2002.
3. M. Paolucci, T. Kawamura, T. R. Payne, K. Sycara, "Importing the Semantic Web in UDDI", Proceedings of E-Services and the Semantic Web Workshop (ESSW 2002), 2002.
4. M. H. Nodine, J. Fowler, T. Ksiezyk, B. Perry, M. Taylor, and A. Unruh, "Active information gathering in infosleuth", International Journal of Cooperative Information Systems, pp. 3–28, 2000.
5. L. Li and I. Horrocks, "A software framework for matchmaking based on semantic web technology", Proceedings of the Twelfth International World Wide Web Conference (WWW 2003), pp. 331–339, 2003.
6. "XML Pointer Language",http://www.w3.org/TR/xptr.
7. "The Rule Markup Initiative", http://www.dfki.uni-kl.de/ruleml/.
8. "Resource Description Framework", http://www.w3.org/RDF/.
9. "DAML Language", http://www.daml.org/language/.
10. "Web-Ontology Working Group", http://www.w3.org/2001/sw/WebOnt/.
11. "DAML Services", http://www.daml.org/services/.
12. "Six Sigma Academy", http://www.6-sigma.com/.