A Framework for Semantic Web Mining Model

1G Ramu, 2Dr B Eswara Reddy
1M.Tech (Computer Science), 2Associate Professor & HOD
2Department of CSE,
JNTUACE, Anantapur, A.P.
E-mail: g.ramucse@gmail.com, eswarcsejntua@gmail.com

Abstract— The Web is transforming from a Web of data to Web services. This trend is provided to compose the services from existing services. Service mining framework is used to find the web services from existing services. This framework is a seven step process to find services but this model finds web services statically. In this paper, a framework is proposed which is a semantic web service mining model that allows finding services from existing services dynamically by using OWL-S ontologies. This work detects the services based on semantic relevance and this semantic relation can be identified by the ontology analysis. An online search model is proposed to detect the semantic relevant web services on the web.

Keywords— Quality of Service, Ontologies, Semantic Web, Web Service Discovery.

I. INTRODUCTION

The creation, deployment, and use of services that meet the needs of individuals and communities in virtually all areas of human endeavor is one of the hallmarks of civilization. We select suitable service providers based on recommendations from friends, family, a person one knows slightly or experts, or by looking them up in directories (e.g., Yellow-Pages). Such type of human-oriented service sele-cition and utilization serve as motivation for Web service discovery in a Service-Oriented Architecture (SOA). SOA supports a directory in which service providers can advertise their services in a form that enables potential clients to find and access them over the Internet. The notion of Semantic Web services [2] takes us one step closer to interoperability of autonomously developed and deployed Web services, where a software agent or application can dynamically find and bind services without having a priori hard-wired knowledge about how to discover and invoke them. OWL-S [3] is a specific OWL [4] ontology designed to provide a framework for semantically describing such services from several perspectives (e.g., discovery, invocations, compositions). During the development of a service, the abstract procedural concepts provided by OWL-S ontology can be used along with the domain specific OWL ontologies which provide the terms, concepts, and relationships used to describe various service properties (i.e., Inputs, Outputs, Preconditions). In general, ontology-based matchmaking is used to find and access service providers against a specific service request. However, this approach serves from several limitations. In a SOA, individual users or group of users are expected to query for services of interest to them using descriptions that are expressed using terms in their own ontologies. But with proliferation of independently implemented and deployed services, the semantic correspondences between the user ontology on which the user queries are based and the domain ontologies on which the service descriptions are based, are likely to change. Consequently, users ought to be able to specify inter-ontology correspondences to facilitate matchmaking between the service requests and service advertisements. Current approaches for describing services on the Semantic Web (e.g., OWL-S [3]) do not support for establishing semantic correspondences between ontologies. Although lately, new frameworks such as, Web Service Modeling Ontology (WSMO) [5] and Web Service Description Language (WSDL-S) [6], have been proposed to provide support for the needed inter-ontology translation. Existing state-of-the-art technologies for publishing and discovering web services (e.g., WSDL [7], UDDI [8]) use static descriptions of service interfaces. Consequently, they lack support for service selection based on non-functional attributes such as Quality of Service (QoS). Some approaches incorporation of QoS criteria in service discovery lack support for dealing with semantic differences among independently developed service specifications [9]. Finally, with the proliferation of Web services and service providers, it is inevitable that there will be services ordered by multiple providers with the same functionality. In such scenarios, the users should be able to rank (or order) the discovered services based on some criteria (e.g., quality of service (QoS) ratings, cost, etc). However, existing approaches for service selection make no provision for user specified ranking criteria as part of the service request. Against this background, this paper builds on the recent developments on Semantic Web services [2] and ontology-based solutions for service selection to develop an approach for discovery of Semantic Web services. In particular, we allow the users to specify context-specific semantic
correspondences between multiple ontologies to resolve semantic differences between them. These correspondences are used for selecting services based on the user’s functional requirements and non-functional requirements, which are then ranked based on a user-specified criteria.

The rest of the paper is structured as follows. Section II describes the related work. Web service discovery, prototype implementation is presented in the section III. The experimental results are presented in section IV. The final section summarizes the conclusion.

II. RELATED WORK

We envision a web wide infrastructure for web services supported by a group of registries that function as directories. These registries record advertisements of services that come online and support search of services that give a set of requested functionalities.

An advertisement matches a request, when the advertisement describes a service that is sufficiently similar to the service requested. Of course, the problem is to specify what “sufficiently similar”. It means in its strongest interpretation, an advertisement and a request are “sufficiently similar” when they describe exactly the same services. This definition is too restrictive, because advertisers and requesters have no prior agreement on how a service is represented; furthermore, they have very different objectives. A restrictive criterion on Semantic Matching of Web Services matching is therefore bound to fail to recognize similarities between advertisements and requests.

To accommodate a soft definition of “sufficiently similar” we need to allow matching engines to perform flexible matches, i.e. matches that recognize the degree of similarity between advertisements and requests. Service requesters should also be allowed to decide the degree of flexibility that they grant to the system. If they concede little flexibility, they reduce the likelihood of finding services that match their requirements, i.e. they minimize the false positive, while increasing the false negative. On the other hand, by increasing the flexibility of match, they achieve the opposite effect: they reduce the false negative at the expense of an increase of false positive.

An additional problem related with performing flexible matches is that the Matching Engine is open to exploitation from advertisements and requests that are too generic in the attempt to maximize the likelihood of matching. For instance, a service may advertise itself as a provider of every-thing, rather than to be honest and precise with what it does. Similarly, a requester may ask for any service, rather than specifying exactly what it expects. The matching engine can reduce the efficacy of these exploitation by ranking advertisements on the basis of the degree of match with the requests. In a nutshell, we expect the matching engine to satisfy the following criteria.

- The matching engine should support flexible semantic matching between advertisements and requests on the basis of the ontologies available to the services and the matching engine.
- Despite the flexibility of match, the matching engine should minimize false positives and false negatives. Furthermore, the requesting service should have some control on the amount of matching flexibility it allows to the system.
- The matching engine should encourage advertisers and requesters to be honest with their descriptions at the cost of paying the price of either not to be matched or being matched inappropriately.
- The matching process should be efficient: it should not burden the requester with excessive delays that would prevent its effectiveness.

III. WEB SERVICES DISCOVERY

A simple architecture of our prototype implementation for discovery of Web services over the Semantic Web is shown fig. 1. Initially, the Service Providers advertise their services (namely, profile, process, grounding in OWL-S [3] terminology) with the Service Registry. This registry serves as a repository for the service advertisements, against which the service request queries are matched. At the time of registration, the Service Registering API parses the OWL-S description (by using Jena [10]) and converts an OWL ontology into a collection of JESS [11] facts, which are stored as triples (i.e., < Subject, Predicate, Object >) in the JESS KB. The JESS reasoning engine can infer more facts to ensure that all the < S, P, O > triples implied by the ontology are stored as facts in JESS KB. The Service Registering API also translates pre-conditions and conditions for outputs and effect in the service description ontology into JESS rules, which are stored in the JESS KB. Typically, the JESS rules Can be considered to be analogous to the conditional if...then statements used in various programming languages. This is because a JESS rule consists of a conditional expression, and a series of commands to execute when that expression is satisfied. The conditional expression occurs on the Left-Hand-Side (LHS) of a rule, whereas, the set of commands to be executed occur on the Right-Hand-Side (RHS). Once all the JESS facts and rules for the service advertisements are stored in the JESS KB, they are evaluated during the matchmaking process against the service request. The Service Requester specifies a request for service selection using the Service Requesting API. Such a request is described using OWL-S. The requester also specifies the
interoperation constraints (ICs) between the terms and concepts of its ontologies to the domain ontologies. These ontologies along with the set of ICs are stored in the Ontology Database. For our first prototype, the constraints are defined manually. However, we are working towards semi automatic approaches for specifying such correspondences [10].

The requester also specifies the interoperation constraints (ICs) between the terms and concepts of its ontologies to the domain ontologies. These ontologies along with the set of ICs are stored in the Ontology Database. For our first prototype, the constraints are defined manually.

However, we are working towards semi automatic approaches for specifying such correspondences [10]. With the help of these translations, the service requesting API transforms the requester’s query, into a domain-specific query. In other words, the API transforms the original service request description (using the terms and concepts from the user ontology) into a pseudo description (using the terms and concepts from the domain ontologies). This description is also translated into JESS facts and rules. The matchmaking engine then tries to find service advertisement which match the user’s request. The matchmaking algorithm that we implemented is based on [14]. This algorithm typically uses subsumption reasoning to find similarity between service advertisements with the requests based on the match between inputs and outputs. Each of these matches are individually scored and the results aggregated to determine a set of candidate service providers, which are then categorized based on their degree of match. These candidate service providers (for each category) are further refined based on whether they satisfy the non-functional requirements of the requester and then ranked on some user-specified ranking criteria (if any), e.g., physical distance between the requester and the service. Finally, the user selects a service provider using his/her prudence.

The main control loop of the matching algorithm is shown in algorithm 1. Requests are matched against all the advertisements stored by the registry. When ever a match between the request and any of the advertisement is found, it is recorded and scored to find the matches with the highest degree.

**Algorithm 1. Main Control Loop**
1. match(request)
2. recordMatch = empty list
3. for all adv in advertisements do
4.   if match(request, adv) then
5.     recordMatch.append(request, adv)
6.   end if
7. return sort(recordMatch);
8. end for

A match between an advertisement and request consist of the match of all the outputs of the request against the outputs of the advertisement; and all the inputs of the advertisement against the inputs of the request. The algorithm for output matching is described in detail in algorithm 2: a match is recognized if and only if for each output of the request, there is a matching output in the advertisements. The degree of success depends on the degree of match detected. If one of the request’s output is not matched by any of the advertisement’s output the match fails. The matching between inputs is computed following the same algorithm, but with the order of the request and the advertisement reversed whereas the request’s outputs are matched against the advertisement’s inputs, the advertisement’s inputs are matched against the request’s inputs.

![Fig.1. Framework for Semantic Web Services Discovery](image)
The degree of match between two outputs or inputs depends on the relation between the concepts associated with those inputs and outputs. We differentiate between four degrees of matching according to the rule displayed in algorithm 3. The rational for the degree assignment is described in algorithm 3.

Algorithm 3. Output Matching
1. degreeOfMatch(outR,outA):
   1. if outA=outR then return exact
   2. If outR subclassOf outA then return exact
   3. If outA subsumes outR then return plugIn
   4. If outR subsumes outA then return subsumes
   5. otherwise fail

   exact if outR=outA then outR and outA are equivalent, which we label as exact. The second clause is a bit more complicated; if outR subclassOf outA then the result is still exact under the assumption that by advertising outA the provider commits to provide outputs consistent with every immediate subtype of outA.

   plug in If outA subsumes outR3 than outA is a set that includes outR, or, in other words, outA could be plugged in place of outR. For example, the a service that provides vehicles could be of use for another service that expects station wagons.

   subsumes If outR subsumes outA, then the provider does not completely fulfill the request. The requester may use the provider to achieve its goals, but it likely needs to modify its plan or perform other requests to complete its task.

   fail Failure occurs when no subsumption relation between advertisement and request is identified.

The last piece of the algorithm to discuss is the scoring system used to sort the resulting matches. Therefore the exact match defined is also based on the subsumption relation. The rules for plug in matching apply when the concepts are not the same and no subclass of relation holds. Rationale behind them is that the requester expects first and foremost that the provider achieves the output requested at the highest degree.

Algorithm 4. Rules for the Degree of Match Assignment
1. sortRule(match1,match2)
2. if match1.output > match2.output then match1 > match2
3. if match1.output = match2.output
   & match1.input > match2.input then match1 > match2
4. if match1.output = match2.output
   & match1.input = match2.input then match1 = match2

This is reflected in our rules by establishing that the main sorting criteria are to select the match with the highest score in the outputs are shown in algorithm 4. Input matching is used only as secondary score to break ties between equally scoring outputs: the requester may solve any mismatch between the information that it has available and the expectations of the provider with additional problem solving or by querying the registry to find additional providers.

IV. EXPERIMENTAL RESULTS
The processing time of an advertisement is measured by calculating the time difference between the UDDI registry receives an advertisement and the time is deliver the results, to eliminate the network latency time.

A. Performance – Publishing Time
In first experiment the time taken to publish an advertisement in an OWL-S/UDDI registry and in a UDDI registry is compared. It’s assumed that the ontologies required by the inputs and outputs of the advertisements are already present in the OWLS/UDDI registry. The advertisements may have different inputs and outputs but they are present in one ontology file, hence the ontology has to be loaded only once, however the registry still has to load 800 advertisements. Table-I shows the average time taken to publish 800 advertisements in a OWL-S/UDDI registry and an UDDI registry. It can be seen that the OWLS/UDDI registry spends around 6-7 times more in time, however since publishing is a one-time event is not concerned about the time taken. For a more detailed analysis of publishing time refer to [16].

B. Performance – Querying Time
In final experiment the time required to process a query is calculated. The queries which are being used do not load any new ontologies into the matchmaker, but they use the ontologies that are already present in the matchmaker. Around 250 queries are used with three inputs and one output. Table-II shows the average time required to process these queries. The standard deviation shows that the time required to process the queries is almost constant. It is does not comparing the query performance of the matchmaker with the standard UDDI because the implementation uses CPU memory to store all the information as opposed to databases. The average query response for the standard UDDI is around 400ms which includes the data base latency.

Table I: Publishing Time without loading ontologies

<table>
<thead>
<tr>
<th>Registry</th>
<th>Time in ms</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDDI</td>
<td>163.98</td>
<td>86.17</td>
</tr>
<tr>
<td>OWL-S/UDDI</td>
<td>1050.77</td>
<td>167.96</td>
</tr>
</tbody>
</table>

Table II: Query processing time

<table>
<thead>
<tr>
<th>Registry</th>
<th>Time in ms</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OWL-S/UDDI</td>
<td>1.306</td>
<td>.54</td>
</tr>
</tbody>
</table>
V. CONCLUSION

In this paper, the challenges posed by existing web service standards to automatically discover and interact with web services are described. The advantages of OWL-S over existing standards are discussed. The difficulties during the development and the consumption processes of OWL-S based web services and how OWL-S IDE supports a developer through this process are discussed. Mainly concentrated on the support of discovery in OWL-S IDE and changes to the existing UDDI registry. Conducted some preliminary experiments to show the scalability of framework implementation. The techniques proposed in this work provide algorithms for the efficient use of OWL-S ontologies in UDDI, it can be easily applied to any OWL ontology. In this sense, the algorithms provided in this paper may provide a valuable basis for an efficient and scalable implementation of the proposed semantic search in UDDI.

ACKNOWLEDGMENT

The authors would like to express their cordial thanks to their esteemed patrons Prof. A. Ananda Rao, Principal, JNTUACE, Anantapur, Prof.V.Sankar, Registrar, JNTU Anantapur and Prof.B. Krishna Gandhi, Vice-Chancellor, JNTU Anantapur, Andhra Pradesh

REFERENCES