Constraint-based Web Services Discovery and Composition

Fabio Gomes de Andrade*, Ulrich Schiel†, Cláudio de Souza Baptista*

*Centro Federal de Educação Tecnológica da Paraíba - Brazil
†Universidade Federal de Campina Grande - Brazil
fabio@cefetpb.edu.br, (ulrich, baptista)@dsc.ufcg.edu.br

Abstract

Recently, the volume of Web information and the number of user accesses have increased rapidly. Static information can be retrieved by search engines whereas dynamic information may need special programs called Web services. Since each web service is specialized on its source, the attendance of complex user necessities requires the composition of several services. Thus, we need an efficient way of both locating the best services on the Web, and composing them adequately in order to solve complex problems. This paper presents a tool that discovers web services based on ontologies. It also realizes sequential and parallel composition and optimizes the result set based on domain specific attributes of service providers.

Index Terms—Web Services, ontologies, web service composition

1. Introduction

In the last years the Web Service platform [1] has gained great acceptance as a way to standardize services published on the web. There are standards on how to publish, describe interfaces and send messages. These standards are XML-based such as the Simple Object Access Protocol (SOAP), the Web Service Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI). SOAP specifies how Web services may encapsulate messages in XML documents. WSDL aims to describe service interfaces offered by Web services. WSDL may describe the service operations, input and output parameters of each message, parameter datatypes and so on. UDDI consists of a framework which enables service publication and discovery. Service providers use UDDI to announce their services; whereas clients use it in order to find out a specific service, and how to interact with it.

Nonetheless, UDDI has strong limitations when concerning efficient and precise retrieval of services. The main reason is that it does not provide any semantics on published services, since only simple textual service description is stored.

In order to overcome this limitation, elements of the Semantic Web have been considered for the description and retrieval of web services. Ontologies offer a structural description of the concepts of a specific domain of knowledge and its relationships. The integration of Semantic Web concepts to web services has been called Semantic Web Services[2].

Several technologies have been proposed for the description of ontologies and semantic markup of services. For the description of ontologies the most representative languages are DAML+OIL and OWL. The semantics of services are usually described using DAML-S, OWL-S or WSMO. Other approaches use Description Logic for both. However at the moment it is not clear which of them will become a standard.

The use of Semantic Web Services has enabled the development of more powerful service location tools. In order to attend a complex user service requirement the best solution should be, in many cases, the composition of several services [3, 4]. This occurs if a unique service could not fulfill completely the requirements. The necessity of composition can be defined previously at design time or can be discovered automatically at runtime.

Recently several solutions have been proposed for the automatic composition of web services [4, 5]. Most of them propose algorithms based on semantic features of each service, mainly its input, output, pre-conditions and events. Some also use non-functional requirements, such as delays, confidence or costs.

If two (composite) or more services match the basic user requirements, additional constraints may be decisive in order to choose the best service. For instance, if we search for lodging, among the matching services the best result may be determined by additional attributes like the existence of a restaurant, a swimming pool, etc.
This paper presents WebS Composer, a tool that allows the location, composition and selection of the best web services considering user attributes and constraints. These constraints determine important features of the provider, and the required input and output parameters.

The tool uses semantic annotation and information retrieval techniques in order to achieve the automatic discovery of composed web services at execution time. Apart from automatic web service discovery, the tool also allows for web service registry by using UDDI and discovery of single services. Moreover, service composition defined at design time is also possible. Finally, the proposed tool manages the running of services.

The main contributions of the paper are:
- a methodology for the automatic detection and composition of services, considering partial matching and ranking;
- best selection of the services based on user constraints;
- optimal sequence composition with a back- and forward chaining algorithm.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 focuses on the main issues of the proposed solution, and highlights semantic annotation. Section 4 discusses our searching strategies. Section 5 presents a performance evaluation of WebS Composer. Finally, section 6 concludes the paper.

2. Related Work

In the last few years, many research works have been focused on web service discovery and automatic composition. Some of these works [6, 7, 8, 9] describe frameworks which enables the specification of service compositions but they lack in proposing an approach for automatic composition.

Other works [5, 10] propose algorithms for automatic compositions, but they focus only on sequential composition, which limits service composition discovery.

Costa et al. [11], presents a solution for both sequential and parallel compositions, however there is no ranking for the discovered compositions. Although Aversano et.al [12] is similar to our approach, as sequential and parallel compositions are automatically discovered, and there is a ranking for them; the proposed solution is complex and lacks flexibility.

Some research work address user constraints to select web services. McIlraith et al. [2] enable users to utilize service outputs to define the execution plan in the composition. Pathak et al. [13] provide to users the possibility of choosing the services by using constraints in the non-functional properties (e.g. cost). Aggrawal et al. [14] enable the users to select services through constraints based on the relationships among service providers. Nonetheless, to the best of our knowledge, none of the existent works use user constraint as a selection criteria with semantic annotation in the service providers. Also, the back and forward algorithm has not been addressed yet.

3. WebS Composer

WebS Composer is a tool which allows the discovery, automatic composition and execution of web services. New service providers are stored in a local database with semantic annotation on the corresponding ontology.

The main characteristics of WebS Composer are:
- **automatic sequential and parallel service composition:** the tool detects if a composition is needed in order to fulfill the query requirements. An execution plan defines the order of service execution;
- **semantic description:** each service is semantically described through a service ontology using OWL-S. This feature aims to improve service discovery;
- **flexible architecture:** new domains may be added and removed without any code modification;
- **service constraints:** besides the parameters that determine the required service, the user can state additional constraints in order to improve the selection of the best service;
- **partial matching:** if a service does not match exactly with the query, it may nevertheless be retrieved so far as it has an acceptable similarity;
- **ranking:** a ranking algorithm considering service constraints and partial matching facilitates user’s final selection of the most adequate (composed) service.

Fig. 1 presents the basic architecture of WebS Composer. It is divided into three tiers: presentation, business logic and data structures.
The user interface module is responsible for providing user-system interaction. Basically, the module receives user requests from a web interface and sends the required functionality to the query module. The result set is then presented to the user which can start the execution of the services.

In the business logic tier, the registry module is responsible for insertion of new services and service provider metadata into the system. The semantic annotation of the new service is stored in a local OWL ontology, corresponding to the specific domain, as an XML document. The corresponding UDDI catalogue is updated accordingly.

The query module is responsible for service localization and composition. It is able to create three types of services: atomic services, static composition, and dynamic composition. When a candidate service composition is found, the module generates an execution plan described in OWL-S. The plan determines the schedule of sequential and parallel service execution.

The execution module is responsible for the execution of the services discovered in the query module. If the service is atomic, it contains a reference to the service to be called. Whereas when the service is composed, the execution plan is realized using the input parameters received from the user interface module. After the service execution, the result set is forwarded to the user interface module in order to be exhibited to the user.

### 3.1. The semantic Markup

The semantic markup of services is used to provide a semantic enhancement to service description. By using such semantics, the query module may provide more precise searches and execute some reasoning.

In order to make the semantic description we consider domain ontologies expressed in OWL. These ontologies have an important role as they define the concepts used in the service input and output. These ontologies are stored in a local database.

By using the ontologies we obtain the OWL-S file which corresponds to the service semantic markup. This file is generated in a simple way. When a user inserts a new service he provides general information (e.g., name, description, etc.) and the service input and output. This information, instead of being based on single data types such as integer, string, or character, is provided using the concepts defined in a given ontology. After that, a parser generates the OWL-S based on the received information. Hence, this file is stored in the local database in XML format. Currently we use a XML type in a Oracle 10G database system.

Notice that besides generating semantic annotation for each service, WebS-Composer also generates semantic annotation for the service provider. When a service provider is registered in the system, it fulfills a form which contains metadata related to conventional UDDI data, such as provider name, description, telephone, email, etc. Moreover, it is also possible to provide properties related to the application domain. These properties come from the domain ontology which represents the kind of enterprise. For example, in a hotel, it might be interesting to have information about parking, number of stars, restaurant, etc. The system will create an instance of the appropriate concept to represent a given enterprise. The aim of this tagging is to use synonyms, specialization, generalization, composition, and inferences in the ontologies in order to select the service provider which are more suitable for the user requirements.

Moreover information obtained from the corresponding domain ontology is inserted. After that, the service provider metadata is also stored in the local database. Also a UDDI registry is created. Hence, the system uses the expressivity power and inference of ontologies in order to improve service discovery.

### 4. Service Searching

In this section we describe the semantic markup structure used for service discovery and composition. Moreover, we discuss how WebS Composer selects the services among several ones retrieved in the result set.

WebS Composer offers two types of queries: simple and advanced. Simple queries may happen in two ways. The first one, occurs when user searches only for stored services which offer a given function, without constraints on the service provider. In the second way, user searches for service providers which have some
features independent of the online services they provide. This can be used when users do not intend to proceed with an electronic transaction, for security reasons, for example. But they are interested in finding service providers with specific features. For instance, a user is planning to travel to a given city and he/she may want to know which hotels have swimming pool and gym, and whether they deal with a given credit card. Thus, there is no concern about the web services offered. In this example, the system looks for hotels which fulfill user constraints, and retrieve them with associated metadata. Notice that current search engines do not solve this kind of query appropriately, as only keywords and logical operators are used.

On the other hand, the advanced query combines the two types of simple queries described previously. It is realized in several steps. Firstly, there is the service filtering according to the user constraints on service providers. Then the automatic service discovery and composition is realized. Finally, the services are executed. We detail further the first two steps.

### 4.1. Filter based on user constraints

In order to find out services that match user needs we use the classic vector model from information retrieval [15]. To use the service filtering based on user constraints, users must inform in their request which are the desirable features of the service provider, and they also pose some constraints on service providers. For instance, when looking for traveler services, users may request some facilities. These elements are specific attributes of the instances of domain ontologies (hotels, restaurants, museums, etc.). We distinguish between mandatory constraints and desirable constraints.

After receiving that information, WebS Composer generates a unitary vector in which each dimension represents one mandatory constraint and a vector for each service which is going to be considered. Each attended constraint receives the value 1. For example, if a user wants to stay in a five stars hotel which contains a swimming pool and tennis court, the vector which represents this service will have three dimensions, one for each expressed condition.

Hence, for each kind of service, our search engine looks for instances of the domain ontology and retrieve those services which are conforming to the desired constraints. All services whose vector coincides with the query vector are selected as candidates.

Then the tool evaluates whether the services fulfill the desirable requirements posed by the user. If the condition is satisfied, the dimension which represents it receives the value 1, otherwise, it receives 0. The similarity level between service provider and user request is obtained through the cosine of the angle between the vector which represents the service provider and the vector which represents the kind of service. Similarity levels which are lesser than a minimum threshold are discarded.

### 4.2. Matching of service functionalities

After choosing services that attend the basic constraints, we must test if they realize the requested functionalities. The retrieval is realized in the following steps. In a visual interface the user states the services he needs and the results he looks for. The interface guides the user to use only terms which are entries in domain ontologies of the system.

For the given inputs and required outputs the system creates vectors which are matched against corresponding vectors of the services. When a query is submitted the system analyses the services selected in the previous step and creates similarity vectors for the inputs, outputs and a global matching vector.

The following algorithm realizes the matching of the output of a query with the output of a service.

```plaintext
for i = 1 to request.size()
    requestVector[i] = 1;
//for each registered service
for i = 1 to S.size()
    outputVector = createVector(S.numberOfOutouts);
    levelOfMatch(request, S)
    outputVector[i] = levelOfMatch(R.output[i], S.outputs);
// starting the request with the value 1 for each dimension
for i = 1 to request.size()
    requestVector[i] = 1;
// for each registered service
max = levelOfMatch(requestConcept, serviceOutputs[1]);
for i = 2 to serviceOutputs.size()
    if levelOfMatch(requestConcept, serviceOutputs[i]) > max
        max = levelOfMatch(requestConcept, serviceOutputs[i]);
return max;
```

At the beginning of the algorithm, the outputs of the user request are represented by a vector. Each dimension represents one output. Each of these dimensions receives the maximum value 1. The aim of this algorithm is to check whether the available service outputs are enough to satisfy user request.

For each service output the system calculates the level of similarity. This level of similarity is given by the cosine of the angle formed between the two vectors.

The vector which represents the matching between service inputs and user request is found analogously. The only difference is that we invert the roles of request
and service, since we evaluate whether the available inputs in the request are or not sufficient to execute the service. The level of similarity between inputs request and service is also calculated analogously.

With these results between inputs and outputs the global matching is obtained as:
1) Create a two-dimensional vector, with value 1 in each dimension, representing the best input and the best output;
2) Create, for each service, a two-dimensional vector with the two similarities obtained in the previous algorithms;
3) The cosine between the two vectors determines the global similarity;
4) All services with a global similarity greater than a pre-defined threshold are candidates to be used.

4.3. The concept of matching algorithm

The matching between attributes of a query with a service is done using the algorithm of Paolucci et.al.[1]. Note that all concepts considered are given in the ontologies. Let R be a concept of the query and S a concept of the service. The following matching may occur:

- **Exact**: the matching is exact if R and S represent the same concept \( R = S \) or if R is a subclass of S \( R \subseteq S \);
- **Plug-in**: the matching is plug-in if R is a part of S, i.e. the concept represented by S can be plugged in the concept represented by R. In this case, concept R is a property of concept S.
- **Subsumes**: R subsumes S if R is a concept that contains S. In this case, the service with S does not satisfy completely the query. Hence, concept S is a property of concept R.
- **Incompatible**: R and S are incompatible if none of the relations above applies between them.

For each case above a value is assigned to the vector. The possible values are: 1, 0.8, 0.4 and 0, for the exact, plug-in, subsumes and incompatible matchings, respectively. This assignment occurs for the two dimensions considered. The algorithm that determines what case occurs, giving R and S, is the following [16]:

\[
\text{degreeOfMatch}(R, S):
\]
\[
\text{if } (R=S) \text{ or } (R \subseteq S) \text{ then return exact}
\]
\[
\text{if } R \text{ subclassOf } A \text{ then return exact}
\]
\[
\text{if } S \text{ subsumes } R \text{ then return plugIn}
\]
\[
\text{if } R \text{ subsumes } S \text{ then return subsumes}
\]
\[
\text{otherwise return fail}
\]

Let \( O = (o_1, .., o_n) \) be the input vector of the requirements of a user and \( S = (s_1, .., s_n) \) the initially empty result vector. If the set of all concepts in a service is \( SO \), the algorithm that determines the matching of an input is:

\[
\text{for } k=1 \text{ to } n \text{ do}
\]
\[
s_k = \max \{ \text{degreeOfMatch}(o_k, s_i) \}, \text{ for all } s_i \text{ in } SO;
\]
\[
\text{return } S;
\]

The matching vector of the outputs is obtained in a similar way.

4.4. The composition of services

There are cases where the matching of single web services does not reach the desired threshold. In this case we must either join or compose several services in order to attend user needs. We consider two kinds of composition: union and sequence.

Composition based on union occurs when there is no single web service that matches all the required attributes of the user, but two or more services may do the job. In this case each service must attend well a subset of the attributes of the query. The composition-by-union algorithm is the following:

\[
\text{findCompositionByUnion}(R, S)
\]
\[
//for each service
\]
\[
\text{for } i = 1 \text{ to } S.\text{size()}
\]
\[
//verify if the input match degree is higher than threshold
\]
\[
I_m1 = \text{inputMatchLevel}(R, S_i);
\]
\[
\text{if } I_m1 \geq \text{threshold}
\]
\[
I.\text{add}(S_i);
\]
\[
//build a new vector to store the result
\]
\[
V = \text{createVector}(R.\text{numberOfOutputs});
\]
\[
//verify the service wich best satisfies each dimension
\]
\[
\text{for } i = 1 \text{ to } V.\text{size()}
\]
\[
\max = I[1];
\]
\[
\text{for each } li \text{ in I do}
\]
\[
\text{outputVector} = I[i].\text{outputMatchVector};
\]
\[
\text{if } (\text{outputVector}[i] > \max.\text{outputMatchVector}[i])
\]
\[
\max = I[i];
\]
\[
V[i] = \max;
\]
\[
//evaluate the found vector i the composition satisfies the requisition
\]
\[
\text{for } i = 1 \text{ to } V.\text{size()}
\]
\[
\text{if } \text{inputMatchLevel}(R, V[i]) < \text{threshold}
\]
\[
\text{newRequest} = \text{createRequest}(R, R.\text{outputMatchVector}[i]);
\]
\[
\text{sequence} = \text{findCompositionBySequence}(\text{newRequeste}, S, V1);
\]
\[
\text{if } \text{sequence} \neq \text{empty}
\]
\[
V[i] = \text{sequence};
\]
\[
\text{else } V[i] = \text{empty};
\]
\[
\text{if } \text{globalMatchLevel}(R, V) \geq \text{threshold}
\]
\[
\text{return } V;
\]
\[
\text{else return empty};
\]

Where:
- R is a user request;
- S is the set of all services;
• Im1 is the match degree between the request and service inputs;
• I is the set of services for which the input match degree is higher than the minimum threshold;
• V is the vector which stores the discovered union composition. For each dimension one service from I is associated;

Even if we apply composition by union, in some cases user needs could not be satisfied completely. In these cases we look for a composition in sequence. This kind of composition could be a solution if the requirements could not be solved in a one shot way, but a set of services, executed in a certain sequence solves the problem.

In order to find the right sequence of services, we use similarities between outputs and inputs of several services. The system creates what is called a bridge between the input and output parameters of the request. The objective is to cover the bridge by a chain of matching services. In order to reduce retrieval time and improve the algorithm, we walk from the two edges of the bridge with a back- and forward chaining algorithm. This approach may be better understood through an example. Suppose the user request provides an input parameter C1 and an output parameter C7, and there are the following services:

<table>
<thead>
<tr>
<th>Service</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>C1</td>
<td>C2</td>
</tr>
<tr>
<td>S2</td>
<td>C2</td>
<td>C3</td>
</tr>
<tr>
<td>S3</td>
<td>C3</td>
<td>C4</td>
</tr>
<tr>
<td>S4</td>
<td>C4</td>
<td>C5</td>
</tr>
<tr>
<td>S5</td>
<td>C4</td>
<td>C6</td>
</tr>
<tr>
<td>S6</td>
<td>C5</td>
<td>C7</td>
</tr>
<tr>
<td>S7</td>
<td>C6</td>
<td>C7</td>
</tr>
</tbody>
</table>

First it looks for services with input compatibility with the input of the user (in our case S1) and for services with output compatibility with the users output (in our case S6 or S7). The former group is the beginning of the bridge and the later the end. Therefore, after this first step we have the following state: \{S1\} \{bridge\} \{S6, S7\}.

Now it looks for services matching with the leftside of the bridge as input (forward) and other services matching with the rightside as output (backward). The new services retrieved are chained at their respective sides. In our example, after this second step the state will be \{S1\} \{S2\} \{bridge\} \{S4, S5\} \{S6, S7\}.

This procedure is repeated recursively and in our example the service S3 will close the bridge reaching the state \{S1\} \{S2\} \{S3\} \{S4, S5\} \{S6, S7\}.

After finishing the algorithm all possible sequences are presented. In our case they are \{S1, S2, S3, S4, S6\} and \{S1, S2, S3, S5, S7\} ordered by the ranking algorithm.

The system also foresees the case where no single sequence fulfills the needs and combines sequential with parallel composition, giving sequences such as \{S1, S2, S3, S6, S7\}. This feature achieves a higher flexibility for the service composition discovery; besides it gives support to more complex composition discovery.

Each Web service used in WebS Composer was implemented using the Java Web Services Development Pack 1.5 (JWSDP) technology. This technology enables rapid web service development, and it is tailored for multiple software platforms. Still, Java Web Services may be integrated with the Java 2 Enterprise Edition technology, which automatically generates WSDL and the mapping file for the web service. Other advantage of JWSDP is its registry and query API for web services, which enables registering and querying of services in service directories, such as UDDI and ebXML. Finally we used an Oracle database to store the services markup and the framework Jena in order to realize the parsing of the OWL files. Java Server Pages technology has been used to implement the user interface.

5. Performance Evaluation

We have compared the behavior of our back- and forward chaining with conventional forward chaining.

The services used in the test were generated randomly, with hypothetical inputs and outputs. In Fig. 2 we show the difference when the number of services increases up to 10,000 services. We have also done tests in which the number of services was fixed and we increased gradually the length of the sequence to be discovered (from 2 to 10 services). The results where similar to the ones obtained in the previous experiment.
6. Conclusion

This paper introduces WebS Composer, a software tool which uses semantic tags based on OWL-S and service constraints in order to improve the service discovery process for simple and composite services at execution time.

One of the main contributions is the use of service constraints as additional criteria for the best web service selection to attend user requests. This quality enables to improve the inference engine of the ontology for service selection and composition. Inadequate services can be discarded and the most appropriate candidates are retrieved.

As further improvement, we plan to include preconditions and effects in the semantic description of the services. Preconditions define conditions to be satisfied before calling a service properly. The effects define what a service execution is going to cause to the real world. These features will enable more precise service searches. In order to implement these features it is necessary to define adequate rules. We are investigating languages which provide such capability. The SWSL language (Semantic Web Service Language) is one of the best candidates. Moreover, we intend to address formal methods to prove the correctness of the compositions. Also a workflow model should be considered which enables a more efficient and secure execution of complex web services.

7. References