Semantics-based Web Service Composition engine

Srividya Kona, Ajay Bansal, Gopal Gupta
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75083

Thomas D. Hite
Metallect Corp.
2400 Dallas Parkway
Plano, TX 75093

Abstract

Service-oriented computing is gaining wider acceptance. We need an infrastructure that allows users and applications to discover, deploy, compose and synthesize services automatically. In this paper we present an approach for automatic service discovery and composition based on semantic description of Web services. The implementation will be used for the WS-Challenge 2007 [1].

1. Introduction

In order to make services ubiquitously available, we need a semantics-based approach such that applications can reason about a service's capability to a level of detail that permits their discovery, deployment, composition and synthesis [6]. Informally, a service is characterized by its input parameters, the outputs it produces, and the side-effect(s) it may cause. The input parameter may be further subject to some pre-conditions, and likewise, the outputs produced may have to satisfy certain post-conditions. For discovery and composition, one could take the syntactic approach in which the services being sought in response to a query simply have their inputs syntactically match those of the query. Alternatively, one could take the semantic approach in which the semantics of inputs and outputs, as well as a semantic description of the side-effect is considered in the matching process. Several efforts are underway to build an infrastructure for service discovery, composition, etc. These efforts include approaches based on the semantic web (such as USDL [4], OWL-S [7], WSML [8], WSDL-S [9]) as well as those based on XML, such as Web Services Description Language (WSDL [5]). Approaches such as WSDL are purely syntactic in nature, that is, they only address the syntactical aspects of a Web service. In this paper we present our approach for automatic service composition which is an extension of our implementation that we used at WS-Challenge 2006 [3].

In section 2 we present the formal definition of the Composition problem. We describe our Service Composition algorithm in section 3. Section 4 presents the design of our software with brief descriptions of the different components of the system followed by conclusions and references.

2. Automated Web service Discovery and Composition

Discovery and Composition are two important tasks related to Web services. In this section we formally describe these tasks. We also develop the requirements of an ideal Discovery/Composition engine.

2.1. The Discovery Problem

Given a repository of Web services, and a query requesting a service (we refer to it as the *query service* in the rest of the text), automatically finding a service from the repository that matches these requirements is the Web service Discovery problem. Valid solutions to the query satisfy the following conditions: (i) they produce at least the query output parameters and satisfy the query post-conditions; (ii) they use only from the provided input parameters and satisfy the query pre-conditions; (iii) they produce the query side-effects. Some of the solutions may be over-qualified, but they are still considered valid as long as they fulfill input and output parameters, pre/post conditions, and side-effects requirements.

2.2. The Composition Problem

Given a repository of service descriptions, and a query with the requirements of the requested service, if a matching service is not found, then the composition task can be performed. The composition problem involves automatically finding a directed acyclic graph of services that can be composed to obtain the desired service. Figure 1 shows an example composite service made up of five services $\mathcal{S}_1$ to $\mathcal{S}_5$. In the figure, I' and CI' are the query input parameters and pre-conditions respectively. O' and CO' are the query

output parameters and post-conditions respectively. Informally, the directed arc between nodes $\mathcal{S}_i$ and $\mathcal{S}_j$ indicates that outputs of $\mathcal{S}_i$ constitute (some of) the inputs of $\mathcal{S}_j$.

Discovery and composition can be viewed as a single problem. Discovery is a simple case of composition where the number of services involved in composition is exactly equal to one.

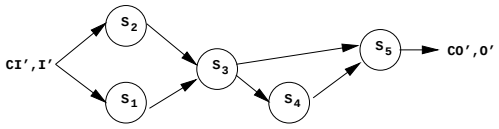

Figure 1. Example of a Composite Service as a Directed Acyclic Graph

Definition (Service): A service is a 6-tuple of its pre-conditions, inputs, side-effect, affected object, outputs and post-conditions. $\mathcal{S} = (\mathcal{CI}, \mathcal{I}, \mathcal{A}, \mathcal{AO}, \mathcal{O}, \mathcal{CO})$ is the representation of a service where $\mathcal{CI}$ is the pre-conditions, $\mathcal{I}$ is the input list, $\mathcal{A}$ is the service's side-effect, $\mathcal{AO}$ is the affected object, $\mathcal{O}$ is the output list, and $\mathcal{CO}$ is the post-conditions.

Definition (Repository of Services): Repository is a set of Web services.

Definition (Query): The *query service* is defined as $\mathcal{Q} = (\mathcal{CI}', \mathcal{I}', \mathcal{A}', \mathcal{AO}', \mathcal{O}', \mathcal{CO}')$ where $\mathcal{CI}'$ is the pre-conditions, $\mathcal{I}'$ is the input list, $\mathcal{A}'$ is the service affect, $\mathcal{AO}'$ is the affected object, $\mathcal{O}'$ is the output list, and $\mathcal{CO}'$ is the post-conditions. These are all the parameters of the requested service.

Definition (Composition): The Composition problem can be defined as automatically finding a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of services from repository $\mathcal{R}$, given query $\mathcal{Q} = (\mathcal{CI}', \mathcal{I}', \mathcal{A}', \mathcal{AO}', \mathcal{O}', \mathcal{CO}')$, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges of the graph. Each vertex in the graph represents a service in the composition. Each outgoing edge of a node (service) represents the outputs and post-conditions produced by the service. Each incoming edge of a node represents the inputs and pre-conditions of the service. The following conditions should hold on the nodes of the graph:

1. $\forall i\ \mathcal{S}_i \in \mathcal{V}$ where $\mathcal{S}_i$ has zero incoming edges, $\mathcal{I}' \sqsupseteq \bigcup_i \mathcal{I}_i$, $\mathcal{CI}' \Rightarrow \wedge_i \mathcal{CI}_i$.
2. $\forall i\ \mathcal{S}_i \in \mathcal{V}$ where $\mathcal{S}_i$ has zero outgoing edges, $\mathcal{O}' \sqsubseteq \bigcup_i \mathcal{O}_i$, $\mathcal{CO}' \Leftarrow \wedge_i \mathcal{CO}_i$.
3. $\forall i\ \mathcal{S}_i \in \mathcal{V}$ where $\mathcal{S}_i$ has at least one incoming edge, let $\mathcal{S}_{i1}$, $\mathcal{S}_{i2}$, ..., $\mathcal{S}_{im}$ be the nodes such that there is a directed edge from each of these nodes to $\mathcal{S}_i$. Then $I_i \sqsubseteq \bigcup_k \mathcal{O}_{ik} \cup I'$, $CI_i \Leftarrow (\mathcal{CO}_{i1} \wedge \mathcal{CO}_{i2} \ldots \wedge \mathcal{CO}_{im} \wedge CI')$.

The meaning of the $\sqsubseteq$ is the subsumption (subsumes) relation and $\Rightarrow$ is the implication relation. Figure 2 explains one instance of the composition problem pictorially. When the number of nodes in the graph is equal to one, the composition problem reduces to the discovery problem. When all nodes in the graph have not more than one incoming edge and not more than one outgoing edge, the problem reduces to a sequential composition problem.

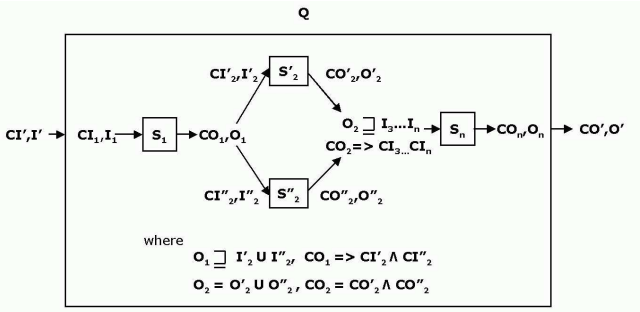

Figure 2. Composite Service

2.3 Requirements of an ideal Engine

The features of an ideal Discovery/Composition engine are:

Correctness: One of the most important requirement for an ideal engine is to produce correct results, i.e, the services discovered and composed by it should satisfy all the requirements of the query. Also, the engine should be able to find all services that satisfy the query requirements.

Small Query Execution Time: Querying a repository of services for a requested service should take a reasonable amount of (small) time, i.e., a few milliseconds. Here we assume that the repository of services may be pre-processed (indexing, change in format, etc.) and is ready for querying. In case services are not added incrementally, then time for pre-processing a service repository is a one-time effort that takes considerable amount of time, but gets amortized over a large number of queries.

Incremental Updates: Adding or updating a service to an existing repository of services should take a small amount of time. A good Discovery/Composition engine should not pre-process the entire repository again, rather incrementally update the pre-processed data (indexes, etc.) of the repository for this new service added.

Cost function: If there are costs associated with every service in the repository, then a good Discovery/Composition engine should be able to give results based on requirements (minimize, maximize, etc.) over the costs. We can extend this to services having an attribute vector associated with them and the engine should be able to give results based on maximizing or minimizing functions over this attribute vector.

These requirements have driven the design of our

semantics-based Composition engine described in the following sections.

3. A Multi-step Narrowing Solution

We assume that a directory of services has already been compiled, and that this directory includes semantic descriptions for each service. In this section we describe our Service Composition algorithm.

Service Composition Algorithm: For service composition, the first step is finding the set of composable services. The correct sequence of execution of these services can be determined by the pre-conditions and post-conditions of the individual services. That is, if a subservice $\mathcal{S}_1$ is composed with subservice $\mathcal{S}_2$, then the post-conditions of $\mathcal{S}_1$ must imply the pre-conditions of $\mathcal{S}_2$. The goal is to derive a single solution, which is a directed acyclic graph of services that can be composed together to produce the requested service in the query. Figure 4 shows a pictorial representation of our composition engine.

In order to produce the composite service which is represented by a graph as shown in figure 1, we filter out services that are not useful for the composition at multiple stages. Figure 3 shows the filtering stages for the particular instance shown in figure 1. The composition routine starts with the query input parameters. It finds all those services from the repository which require a subset of the query input parameters. In figure 3, CI, I are the pre-conditions and the input parameters provided by the query. $\mathcal{S}_1$ and $\mathcal{S}_2$ are the services found after step 1. $\mathcal{O}_1$ is the union of all outputs produced by the services at the first stage. For the next stage, the inputs available are the query input parameters and all the outputs produced by the previous stage, i.e., $\mathcal{I}_2$ = $\mathcal{O}_1 \cup I$. $\mathcal{I}_2$ is used to find services at the next stage, i.e., all those services that require a subset of $\mathcal{I}_2$. In order to make sure we do not end up in cycles, we get only those services which require at least one parameter from the outputs produced in the previous stage. This filtering continues until all the query output parameters are produced. At this point we make another pass in the reverse direction to remove redundant services which do not directly or indirectly contribute to the query output parameters. This is done starting with the output parameters working our way backwards.

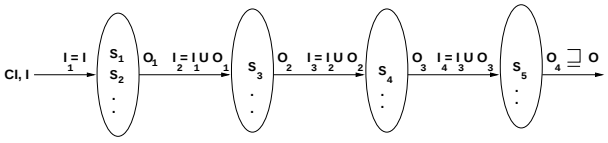

Figure 3. Composite Service

Algorithm: Composition
Input: QI - QueryInputs, QO - QueryOutputs, QCI - Pre-Cond, QCO - Post-Cond
Output: Result - ListOfServices
1. L ← NarrowServiceList(QI, QCI);
2. O ← GetAllOutputParameters(L);
3. CO ← GetAllPostConditions(L);
4. While Not (O ⊒ QO)
5. I = QI ∪ O; CI ← QCI ∧ CO;
6. L' ← NarrowServiceList(I, CI);
7. End While;
8. Result ← RemoveRedundantServices(QO, QCO);
9.Return Result;

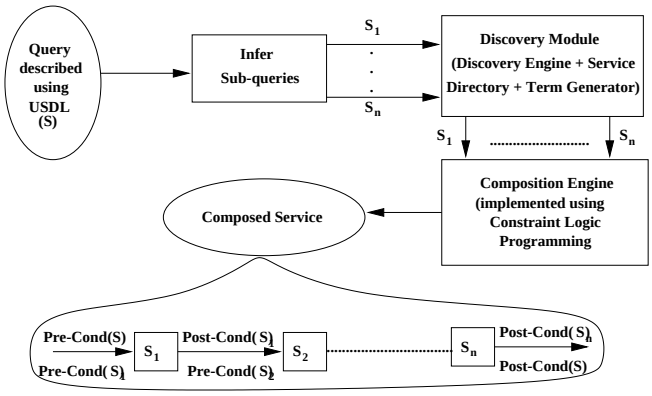

Figure 4. Composition Engine

4. Implementation

Our composition engine is implemented using Prolog [10] with Constraint Logic Programming over finite domain [11], referred to as CLP(FD) hereafter. In this section we briefly describe our software system and its modules. The details of the implementation along with performance results are shown in [2].

Triple Generator: The triple generator module converts each service description into a triple as follow:
(Pre-Conditions, affect-type(affected-object, I, O), Post-Conditions).
The function symbol *affect-type* is the side-effect of the service and *affected object* is the object that changed due to the side-effect. *I* is the list of inputs and *O* is the list of outputs. *Pre-Conditions* are the conditions on the input parameters and *Post-Conditions* are the conditions on the output parameters. Services are converted to triples so that they can be treated as terms in first-order logic. In case conditions on a service are not provided, the *Pre-Conditions* and *Post-Conditions* in the triple will be null. Similarly if the affect-type is not available, this module assigns a generic affect to the service.

Query Reader: This module reads a query file (in XML format, possibly different from the XML format used for a

service) and converts it into a triple used for querying in our engine.

Semantic Relations Generator: We obtain the semantic relations from the provided ontology. This module extracts all the semantic relations and creates a list of Prolog facts.

Composition Query Processor: The composition engine is written using Prolog with CLP(FD) library. It uses a repository of facts, which contains list of services, their input and output parameters and the semantic relations between the parameters. The following is the code snippet of our composition engine:

```
composition(sol(Qname,A)) :-
   dQuery(Qname,_,_),
   minimize(compTask(Qname,A,SeqLen),SeqLen).

compTask(Qname, A, SeqLen) :-
 dQuery(Qname,QI,QO), encodeParam(QO,OL),
 narrowO(OL,SL), fd_set(SL,Sset),
 fdset_member(S_Index,Sset),
 getExtInpList(QI,InpList),
 encodeParam(InpList,IL), list_to_fdset(IL,QIset),
 serv(S_Index,SI,_), list_to_fdset(SI,SIset),
 fdset_subtract(SIset,QIset,Iset),
 comp(QIset,Iset,[S_Index],SA,CompLen),
 SeqLen #= CompLen + 1,  decodeS(SA,A).

comp(_, Iset, A, A, 0) :- empty_fdset(Iset),!.
comp(QIset, Iset, A, SA, SeqLen) :-
 fdset_to_list(Iset,OL),
 narrowO(OL,SL), fd_set(SL,Sset),
 fdset_member(SO_Index,Sset), serv(SO_Index,SI,_),
 list_to_fdset(SI,SIset),
 fdset_subtract(SIset,QIset,DIset),
 comp(QIset,DIset,[SO_Index|A],SA,CompLen),
 SeqLen #= CompLen + 1.
```

The query is converted into a Prolog query that looks as follows:

composition(queryService, ListOfServices).

The engine will try to find a *ListOfServices* that can be composed into the requested *queryService*. Our engine uses the built-in, higher order predicate "bagof" to return all possible *ListOfServices* that can be composed to get the requested *queryService*.

Output Generator: After the Composition Query processor finds a matching service, or the graph of atomic services for a composed service, the results are sent to the output generator in the form of triples. This module generates the output files in any desired XML format. For the WS-Challenge, this module will produce output files in the format provided [1].

For this year's challenge, the software has to receive requests and return results via SOAP. Hence our software will work as a Web service whose interface will accept the discovery/composition query.

5. Conclusion

To catalogue, search and compose services in a semi-automatic to fully-automatic manner we need infrastructure to publish services, document services and query repositories for matching services. We presented our approach for Web service composition. Our composition engine can find a graph of atomic services that can be composed to form the desired service as opposed to simple sequential composition in our previous work [3]. Given semantic description of Web services, our solution produces accurate and quick results. We are able to apply many optimization techniques to our system so that it works efficiently even on large repositories. The use of Constraint Logic Programming (CLP) helped greatly in obtaining an efficient implementation of this system. We used a number of built-in features such as indexing, set operations, and constraints and hence did not have to spend time coding these ourselves. These CLP(FD) built-ins facilitated the fast execution of queries.

References

[1] WS Challenge 2007 `http://ws-challenge.org`.

[2] S. Kona, A. Bansal, G. Gupta, and T. Hite. Efficient Web Service Discovery and Composition using Constraint Logic Programming. In *ALPSWS Workshop at FLoC 2006*.

[3] S. Kona, A. Bansal, G. Gupta, and T. Hite. Web Service Discovery and Composition using USDL. In *CEC/EEE*, June 2006.

[4] A. Bansal, S. Kona, L. Simon, A. Mallya, G. Gupta, and T. Hite. A Universal Service-Semantics Description Language. In *European Conference On Web Services*, pp. 214-225, 2005.

[5] Web Services Description Language. `http://www.w3.org/TR/wsdl`.

[6] S. McIlraith, T.C. Son, H. Zeng. Semantic Web Services. In *IEEE Intelligent Systems Vol. 16, Issue 2, pp. 46-53, March 2001*.

[7] OWL-S `www.daml.org/services/owl-s/1.0/owl-s.html`.

[8] WSML: Web Service Modeling Language. `www.wsmo.org/wsml/`.

[9] WSDL-S: Web Service Semantics. `http://www.w3.org/Submission/WSDL-S`.

[10] L. Sterling and S. Shapiro. The Art of Prolog. *MIT Press, 1994*.

[11] K. Marriott and P. J. Stuckey. Programming with Constraints: An Introduction. *MIT Press, 1998*.