

Semantics-based Efficient Web Service Discovery and Composition

Srividya Kona, Ajay Bansal, and Gopal Gupta¹
and Thomas D. Hite²

¹ Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75083

² Metallec Corp.
2400 Dallas Parkway, Plano, TX 75093

Abstract. Service-oriented computing is gaining wider acceptance. For Web services to become practical, an infrastructure needs to be supported that allows users and applications to discover, deploy, compose and synthesize services automatically. For this automation to be effective, formal semantic descriptions of Web services should be available. In this paper we formally define the Web service discovery and composition problem and present an approach for automatic service discovery and composition based on semantic description of Web services. We also report on an implementation of a semantics-based automated service discovery and composition engine that we have developed. This engine employs a multi-step narrowing algorithm and is efficiently implemented using the constraint logic programming technology. The salient features of our engine are its scalability, i.e., its ability to handle very large service repositories, and its extremely efficient processing times for discovery and composition queries. We evaluate our algorithms for automated discovery and composition on repositories of different sizes and present the results.

1 Introduction

A Web service is a program accessible over the web that may effect some action or change in the world (i.e., causes a side-effect). Examples of such side-effects include a web-base being updated because of a plane reservation made over the Internet, a device being controlled, etc. An important future milestone in the Web's evolution is making *services* ubiquitously available. As automation increases, these Web services will be accessed directly by the applications rather than by humans [8]. In this context, a Web service can be regarded as a "programmable interface" that makes application to application communication possible. An infrastructure that allows users to discover, deploy, synthesize and compose services automatically is needed in order to make Web services more practical.

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 [3]. 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, composition, etc., 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, or, 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 [18–20] for service discovery, composition, etc. These efforts include approaches based on the semantic web (such as USDL [1], OWL-S [4], WSML [5], WSDL-S [6]) as well as those based on XML, such as Web Services Description Language (WSDL [7]). Approaches such as WSDL are purely syntactic in nature, that is, they only address the syntactical aspects of a Web service [14].

Given a formal description of the context in which a service is needed, the service(s) that will precisely fulfill that need can be automatically determined. This task is called discovery. If the service is not found, the directory can be searched for two or more services that can be composed to synthesize the required service. This task is called composition. In this paper we present an approach for automatic discovery and composition of Web services using their semantic descriptions.

Our research makes the following novel contributions: (i) We formally define the discovery and composition problems; to the best of our knowledge, the formal description of the generalized composition problem has been given for the first time; (ii) We present efficient and scalable algorithms for solving the discovery and composition problem that take semantics of services into account; and, (iii) we present a prototype implementation based on constraint logic programming that works efficiently on large repositories.

The rest of the paper is organized as follows. Section 2 describes the two major Web services tasks, namely, discovery and composition with their formal definitions. In section 3 and 4, we present our multi-step narrowing solution and implementation for automatic service discovery and composition. Finally we present our performance results, related work and conclusions.

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. Only those services that produce at least the requested output parameters that satisfy the post-conditions and use only from the provided input parameters that satisfy the pre-conditions and produce the same side-effects can be valid solutions to the query. 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.

Example 1: Say we are looking for a service to buy a book and the directory of services contains services S_1 and S_2 . The table 1 shows the input/output parameters of the query and services S_1 and S_2 .

Service	Input Parameters	Pre-conditions	Output Parameters	Post-Cond
Query	<i>BookTitle, CreditCardNumber, AuthorName, CreditCardType</i>	<i>IsNumeric(CreditCardNumber)</i>	<i>ConfirmationNumber</i>	
S_1	<i>BookName, AuthorName, BookISBN, CreditCardNumber</i>		<i>ConfirmationNumber</i>	
S_2	<i>BookName, CreditCardNumber</i>	<i>IsNumeric(CreditCardNumber)</i>	<i>ConfirmationNumber, TrackingNumber</i>	

Table 1. Example 1

In this example service S_2 satisfies the query, but S_1 does not as it requires *BookISBN* as an input but that is not provided by the query. Our query requires *ConfirmationNumber* as the output and S_2 produces *ConfirmationNumber* and *TrackingNumber*. The extra output produced can be ignored. Also the semantic descriptions of the service input/output parameters should be the same as the query parameters or have the subsumption relation. The discovery engine should be able to infer that the query parameter *BookTitle* and input parameter *BookName* of service S_2 are semantically the same concepts. This can be inferred using semantics from the ontology provided. The query also has a pre-condition that the *CreditCardNumber* is numeric which should logically imply the pre-condition of the discovered service.

Definition (Service): A service is a 6-tuple of its pre-conditions, inputs, side-effect, affected object, outputs and post-conditions. $S = (\mathcal{CI}, \mathcal{I}, \mathcal{A}, \mathcal{AO}, \mathcal{O}, \mathcal{CO})$ 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 $Q = (\mathcal{CI}', \mathcal{I}', \mathcal{A}', \mathcal{AO}', \mathcal{O}', \mathcal{CO}')$ 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 (Discovery): Given a repository \mathcal{R} and a query Q , the Discovery problem can be defined as automatically finding a set \mathcal{S} of services from \mathcal{R} such that $\mathcal{S} = \{s \mid s = (\mathcal{CI}, \mathcal{I}, \mathcal{A}, \mathcal{AO}, \mathcal{O}, \mathcal{CO}), s \in \mathcal{R}, \mathcal{CI}' \Rightarrow \mathcal{CI}, \mathcal{I} \sqsubseteq \mathcal{I}', \mathcal{A} = \mathcal{A}', \mathcal{AO} = \mathcal{AO}', \mathcal{CO} \Rightarrow \mathcal{CO}', \mathcal{O} \sqsupseteq \mathcal{O}'\}$. The meaning of \sqsubseteq is the subsumption (subsumes) relation and \Rightarrow is the implication relation. For example, say x and y are input and output parameters respectively of a service. If a query has $(x > 5)$ as a pre-condition and $(y > -x)$ as post-condition, then a service with pre-condition $(x > 0)$ and post-condition $(y > x)$ can satisfy the query as $(x > 5) \Rightarrow (x > 0)$ and $(y > x) \Rightarrow (y > -x)$ since $(x > 0)$. Figure 1 explains the discovery problem pictorially.

2.2 The Composition Problem

Given a repository of service descriptions, and a query with the requirements of the requested service, in case a matching service is not found, the composition problem

involves automatically finding a directed acyclic graph of services that can be composed to obtain the desired service. Figure 2 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 .

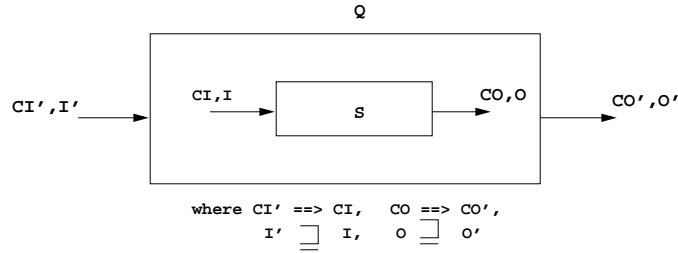


Fig. 1. Substitutable Service

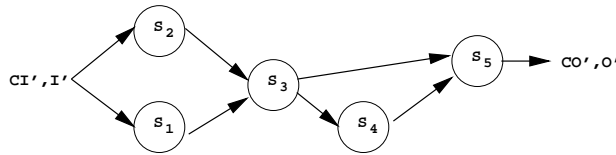


Fig. 2. Example of a Composite Service as a Directed Acyclic Graph

Example 2: Suppose we are looking for a service to buy a book and the directory of services contains services *GetISBN*, *GetAvailability*, *AuthorizeCreditCard*, and *PurchaseBook*. The table 2 shows the input/output parameters of the query and these services.

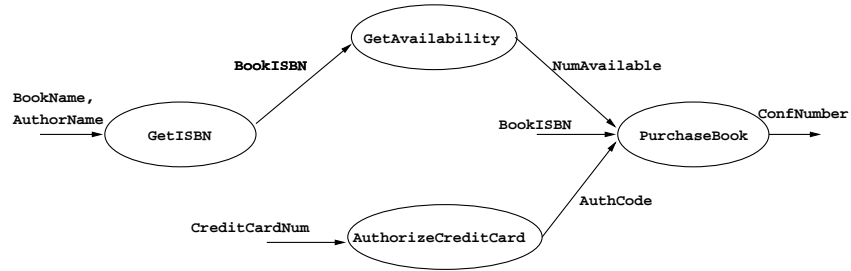


Fig. 3. Example of a Composite Service

Suppose the repository does not find a single service that matches these criteria, then it synthesizes a composite service from among the set of services available in the repository. Figure 3 shows this composite service. The post-conditions of the service *GetAvailability* should logically imply the pre-conditions of service *PurchaseBook*.

Service	Input Parameters	Pre-Conditions	Output Params	Post-Conditions
Query	BookTitle, CreditCardNum AuthorName, CardType		ConfNumber	
GetISBN	BookName, AuthorName		BookISBN	
GetAvailability	BookISBN		NumAvailable	NumAvailable > 0
Authorize CreditCard	CreditCardNum		AuthCode	AuthCode > 99 ∧ AuthCode < 1000
PurchaseBook	BookISBN, NumAvailable, AuthCode	NumAvailable > 0	ConfNumber	

Table 2. Example 2

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

The meaning of the \sqsubseteq is the subsumption (subsumes) relation and \Rightarrow is the implication relation. Figure 4 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.

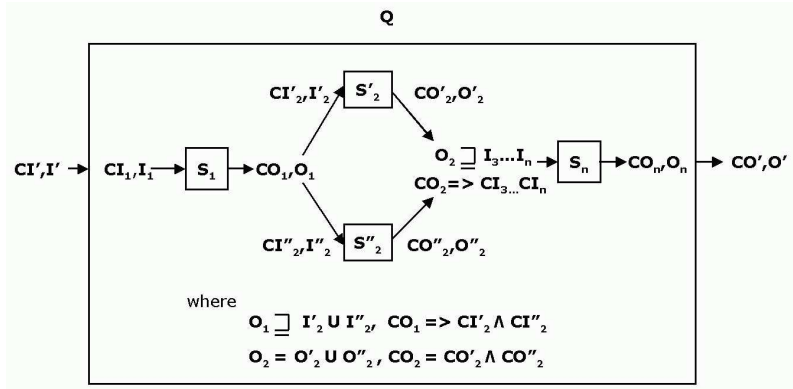


Fig. 4. Composite Service

2.3 Requirements of an ideal Engine

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. 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 Discovery and Composition engine described in the following sections.

2.4 Semantic Description of Web Services

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface that is described in a machine-processible format so that other systems can interact with the Web service through its interface using messages. The automation of Web service tasks (discovery, composition, etc.) can take place effectively only if formal semantic descriptions of Web services are available. Currently, there are a number of approaches for describing the semantics of Web services such as OWL-S [4], WSML [5], WSDL-S [6], and USDL [1].

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 our implementation, we use semantic descriptions written in the language called USDL [1]. The repository of services contains one USDL description document for each service.

USDL is a language that service developers can use to specify formal semantics of Web services. In order to provide semantic descriptions of services, we need an ontology that is somewhat coarse-grained yet universal, and at a similar conceptual level to

common real world concepts. USDL uses WordNet [9] which is a sufficiently comprehensive ontology that meets these criteria. Thus, the “meaning” of input parameters, outputs, and the side-effect induced by the service is given by mapping these syntactic terms to *concepts* in WordNet (see [1] for details of the representation). Inclusion of USDL descriptions, thus makes services directly “semantically” searchable. However, we still need a query language to search this directory, i.e., we need a language to frame the requirements on the service that an application developer is seeking. USDL itself can be used as such a query language. A USDL description of the desired service can be written, a query processor can then search the service directory for a “matching” service. Due to lack of space, we do not go into the details of the language in this paper. They are available in our previous work [2].

With the formal definition of the Discovery and Composition problem, presented in the previous section, one can see that there can be many approaches to solving the problem. Our approach is based on a multi-step narrowing of the list of candidate services using various constraints at each step. In this section we discuss our Discovery and Composition algorithms in detail. These algorithms can be used with other Semantic Web service description languages as well. It will involve extending our implementation to work for other description formats and we are looking into that as part of our future work.

3.1 The Service Discovery Algorithm

The discovery algorithm takes in the query parameters and produces a list of matching services. Our algorithm first uses the query output parameters and post-conditions to narrow down the list of services in the repository. It gets all those services that produce at least the query outputs (i.e., the output parameters provided by a service must be equivalent to or be subsumed by the required output in the query) and whose post-conditions logically imply the query post-conditions. From the list of services obtained, we find the set of all inputs parameters of all services in the list, say I . Then a set of unprovided inputs, say UI is obtained by computing the set difference of I and the query inputs QI . Then the list of services is further narrowed down by removing any service that has even one of the inputs from the set UI . After all such services are removed, we filter our list of services based on query pre-conditions and the required side-effects. The remaining list is our final list of services called *Result*. Figure 5 shows a pictorial representation of our discovery engine.

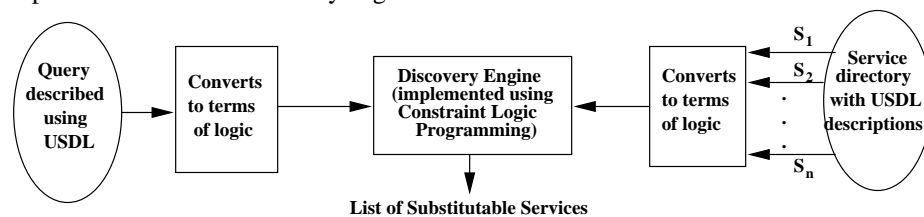


Fig. 5. Discovery Engine

Algorithm: Discovery

Input: QI - QueryInputs, QO - QueryOutputs,

QCI - Pre-Conditions, QCO - Post-Conditions

Output: Result - ListOfServices

1. $L \leftarrow \text{NarrowServiceList}(QO, QCO)$;
2. $I \leftarrow \text{GetAllInputParameters}(L)$;
3. $WI \leftarrow \text{GetUnprovidedInputs}(I, QI)$; i.e., $UI = I - QI$
4. $S \leftarrow \text{FilterServicesWithUnprovidedInputs}(UI, L)$;
5. $\text{Result} \leftarrow \text{FilterServicesWithPreConditions}(QCI, S)$;
6. *Return Result*;

3.2 The Service Composition Algorithm

For service composition, the first step is finding the set of composable services. USDL itself is used to specify the requirements of the composed service that an application developer is seeking. Using the discovery engine, individual services that make up the composed service can be selected. Part substitution techniques [2] can be used to find the different parts of a whole task and the selected services can be composed into one by applying the correct sequence of their execution. The correct sequence of execution can be determined by the pre-conditions and post-conditions of the individual services. That is, if a subservice S_1 is composed with subservice S_2 , then the post-conditions of S_1 must imply the pre-conditions of 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 7 shows a pictorial representation of our composition engine.

In order to produce the composite service which is the graph, as shown in the example figure 2, we filter out services that are not useful for the composition at multiple stages. Figure 6 shows the filtering technique for the particular instance shown in figure 2. 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 6, CI, I are the pre-conditions and the input parameters provided by the query. S_1 and S_2 are the services found after step 1. 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., $I_2 = O_1 \cup I$. I_2 is used to find services at the next stage, i.e., all those services that require a subset of 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.

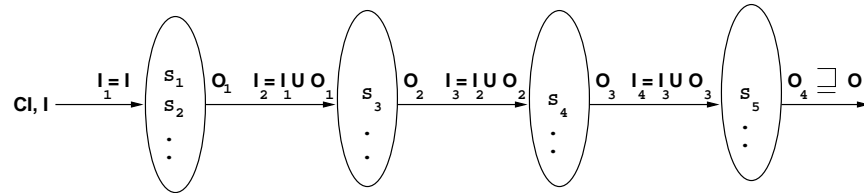


Fig. 6. Composite Service

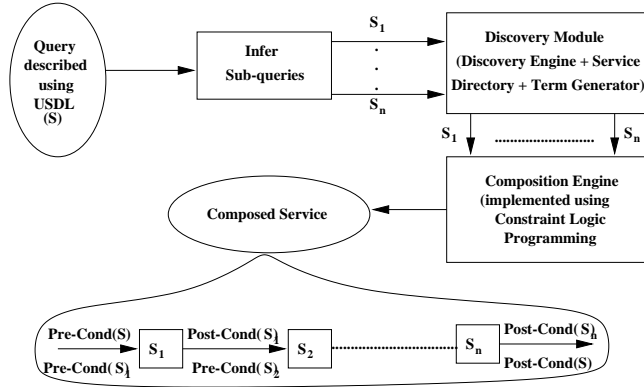


Fig. 7. Composition Engine

Algorithm: Composition

Input: QI - QueryInputs, QO - QueryOutputs, QCI - Pre-Cond, QCO - Post-Cond

Output: Result - ListOfServices

1. $L \leftarrow \text{NarrowServiceList}(QI, QCI)$;
2. $O \leftarrow \text{GetAllOutputParameters}(L)$;
3. $CO \leftarrow \text{GetAllPostConditions}(L)$;
4. While Not ($O \sqsupseteq QO$)
5. $I = QI \cup O$; $CI \leftarrow QCI \wedge CO$;
6. $L' \leftarrow \text{NarrowServiceList}(I, CI)$;
7. End While;
8. Result $\leftarrow \text{RemoveRedundantServices}(QO, QCO)$;
9. Return Result;

4 Implementation

Our discovery and composition engine is implemented using Prolog [11] with Constraint Logic Programming over finite domain [10], referred to as CLP(FD) hereafter. The high-level design of the Discovery and Composition engines is shown in Figure 8. The software system is made up of the following components.

Triple Generator: The triple generator module converts each service description into a triple. In this case, USDL descriptions are converted to triples like:

(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 and specialized unification algorithms can be applied to obtain exact, generic, specific, part and whole substitutions [2]. 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 the query file and passes it on to the Triple Generator. We use USDL itself as the query language. A USDL description of the desired service can be written, which is read by the query reader and converted to a triple. This module can be easily extended to read descriptions written in other languages.

Semantic Relations Generator: We obtain the semantic relations from the OWL WordNet ontology. OWL WordNet ontology provides a number of useful semantic relations like synonyms, antonyms, hyponyms, hypernyms, meronyms, holonyms and many more. USDL descriptions point to OWL WordNet for the meanings of concepts. A theory of service substitution is described in detail in [2] which uses the semantic relations between basic concepts of WordNet, to derive the semantic relations between services. This module extracts all the semantic relations and creates a list of Prolog facts.

Discovery Query Processor: This module compares the discovery query with all the services in the repository. The processor works as follows:

1. On the output parameters of a service, the processor first looks for an *exact* substitutable. If it does not find one, then it looks for a parameter with hyponym relation [2], i.e., a *specific* substitutable.
2. On the input parameters of a service, the processor first looks for an *exact* substitutable. If it does not find one, then it looks for a parameter with hypernym relation [2], i.e., a *generic* substitutable.

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

```
discovery(sol(Qname,A)) :- dQuery(Qname,I,O), encodeParam(O,OL),
/* Narrow candidate services(S) using output list(OL)*/
narrowO(OL,S), fd_set(S,FDs), fdset_to_list(FDs,SL),
/* Expand InputList(I) using semantic relations */
getExtInpList(I, ExtInpList), encodeParam(ExtInpList,IL),
/* Narrow candidate services(SL) using input list (IL)*/
narrowI(IL,SL,SA), decodeS(SA,A).
```

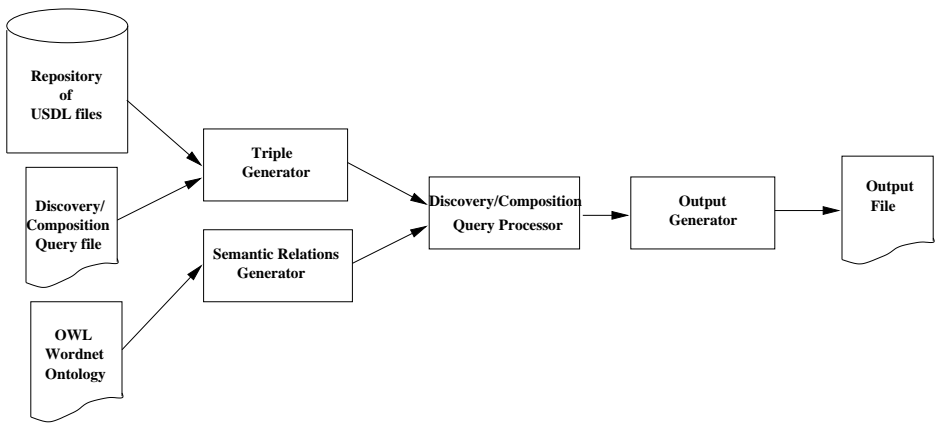


Fig. 8. High-level Design

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

discovery(sol(queryService, ListOfSolutionServices).

The engine will try to find a list of *SolutionServices* that match the *queryService*.

Composition Query Processor: Similar to the discovery engine, composition engine is also 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 Discovery/Composition Query processor finds a matching service, or the list 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.

5 Efficiency and Scalability Issues

In this section we discuss the salient features of our system with respect to the efficiency and scalability issues related to Web service discovery and composition problem. It is because of these features, we decided on the multi-step narrowing based approach to solve these problems and implemented it using constraint logic programming.

Correctness: Our system takes into account all the services that can be satisfied by the provided input parameters and pre-conditions at every step of our narrowing algorithm. So our search space has all the possible solutions. Our backward narrowing step, which removes the redundant services, does so taking into account the output parameters and post-conditions. So our algorithm will always find a correct solution (if one exists) in the minimum possible steps. The formal proof of correctness and minimality is beyond the scope of this paper.

Pre-processing: Our system initially pre-processes the repository and converts all service descriptions into Prolog terms. The semantic relations are also processed and loaded as Prolog terms in memory. Once the pre-processing is done, then discovery or composition queries are run against all these Prolog terms and hence we obtain results quickly and efficiently. The built-in indexing scheme and constraints in CLP(FD) facilitate the fast execution of queries. During the pre-processing phase, we use the term

representations of services to set up constraints on services and the individual input and output parameters. This further helped us in getting optimal results.

Execution Efficiency: The use of CLP(FD) helped significantly in rapidly obtaining answers to the discovery and composition queries. We tabulated processing times for different size repositories and the results are shown in Section 6. As one can see, after pre-processing the repository, our system is quite efficient in processing the query. The query execution time is insignificant.

Programming Efficiency: The use of Constraint Logic Programming helped us in coming up with a simple and elegant code. 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. This made our approach efficient in terms of programming time as well. Not only the whole system is about 200 lines of code, but we also managed to develop it in less than 2 weeks.

Scalability: Our system allows for incremental updates on the repository, i.e., once the pre-processing of a repository is done, adding a new service or updating an existing one will not need re-execution of the entire pre-processing phase. Instead we can easily update the existing list of CLP(FD) terms loaded in the memory and run discovery and composition queries. Our estimate is that this update time will be negligible, perhaps a few milliseconds. With real-world services, it is likely that new services will get added often or updates might be made on existing services. In such a case, avoiding repeated pre-processing of the entire repository will definitely be needed and incremental update will be of great practical use. The efficiency of the incremental update operation makes our system highly scalable.

Use of external Database: In case the repository grows extremely large in size, then saving off results from the pre-processing phase into some external database might be useful. This is part of our future work. With extremely large repositories, holding all the results of pre-processing in the main memory may not be feasible. In such a case we can query a database where all the information is stored. Applying incremental updates to the database is easily possible thus avoiding recomputation of pre-processed data .

Searching for Optimal Solution: If there are any properties with respect to which the solutions can be ranked, then setting up global constraints to get the optimal solution is relatively easy with the constraint based approach. For example, if each service has an associated cost, then the discovery and the composition problem can be redefined to find the solutions with the minimal cost. Our system can be easily extended to take these global constraints into account.

6 Performance

We evaluated our approach on different size repositories and tabulated Pre-processing and Query Execution time. We noticed that there was a significant difference in pre-processing time between the first and subsequent runs (after deleting all the previous pre-processed data) on the same repository. What we found is that the repository was cached after the first run and that explained the difference in the pre-processing time for subsequent runs. We used auto-generated repositories from WS-Challenge website[13], slightly modified to fit into USDL framework. Table 3 shows performance results for

our Discovery Algorithm and table 4 shows results for Composition. The times shown in the tables are the wall clock times. The actual CPU time to pre-process the repository and execute the query should be less than or equal to the wall clock time. The results are plotted in figure 9. The graphs exhibit behavior consistent with our expectations: for a fixed repository size, the preprocessing time increases with the increase in number of input/output parameters. Similarly, for fixed input/output sizes, the preprocessing time is directly proportional to the size of the repository. However, what is surprising is the efficiency of service query processing, which is negligible (just 1 to 3 msecs) even for complex queries with large repositories.

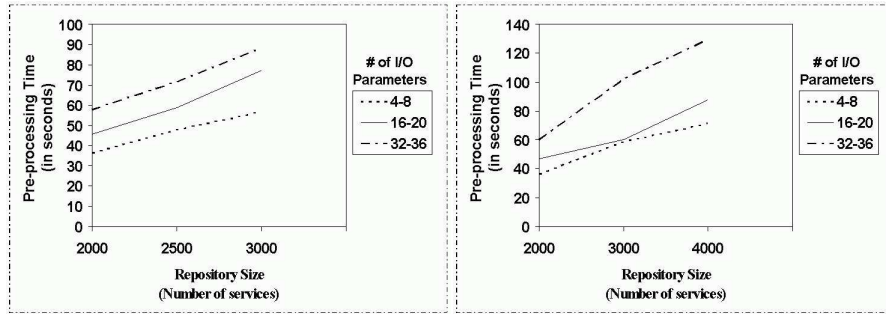


Fig. 9. Performance of Discovery and Composition Algorithm

<i>Repository Size (num of services)</i>	<i>Number of I/O parameters</i>	<i>PreProcessing Time(secs)</i>	<i>QueryExec Time(msecs)</i>	<i>Incremental update (msecs)</i>
2000	4-8	36.5	1	18
2000	16-20	45.8	1	23
2000	32-36	57.8	2	28
2500	4-8	47.7	1	19
2500	16-20	58.7	1	23
2500	32-36	71.6	2	29
3000	4-8	56.8	1	19
3000	16-20	77.1	1	26
3000	32-36	88.2	3	29

Table 3. Performance of Discovery Engine

7 Related Work

Discovery and composition of Web services has been active area of research recently [14, 16, 17]. Most of these approaches are based on capturing the formal semantics of the service using an action description languages or some kind of logic (e.g., description logic). The service composition problem is reduced to a planning problem where the sub-services constitute atomic actions and the overall service desired is represented by the goal to be achieved using some combination of atomic actions. A planner is then used to determine the combination of actions needed to reach the goal. In contrast, we rely more on WordNet (which we use as a universal ontology) and the meronymous

relationships of WordNet lexemes to achieve automatic composition. To the best of our knowledge, most of these approaches that use planning are restricted to sequential compositions, rather than a directed acyclic graph.

<i>Repository Size (num of services)</i>	<i>Number of I/O parameters</i>	<i>PreProcessing Time(secs)</i>	<i>QueryExec Time(msecs)</i>	<i>Incremental update (msecs)</i>
2000	4-8	36.1	1	18
2000	16-20	47.1	1	23
2000	32-36	60.2	1	30
3000	4-8	58.4	1	19
3000	16-20	60.1	1	20
3000	32-36	102.1	1	34
4000	4-8	71.2	1	18
4000	16-20	87.9	1	22
4000	32-36	129.2	1	32

Table 4. Performance of Composition Engine

The approaches proposed by others also rely on a domain specific ontology (specified in OWL/DAML), namely, to discover/compose such services the engine has to be aware of the domain specific ontology. Thus, for these approaches, a completely general discovery and composition engines cannot be built. Additionally, the domain specific ontology has to be quite extensive in that any relationship that can possibly exist between two terms in the ontology must be included in the ontology. In contrast, in our approach, the complex relationships (USDL concepts) that might be used to describe services or their inputs and outputs are part of USDL descriptions and not the ontology. Note that our approach is quite general, and it will work for domain specific ontologies as well, as long as the synonym, antonym, hyponym, hypernym, meronym, and holonym relations are defined between the various terms of the ontology.

A process-level composition solution based on OWL-S is proposed in [21]. This work produces a complete process-level description of a composite service, when the descriptions of the services that are involved in the composition are provided. In contrast, we automatically find the services that are suitable for composition in order to provide the requested service.

Another related area of research involves message conversation constraints, also known as behavioral signatures [15]. Behavior signature models do not stray far from the explicit description of the lexical form of messages, they expect the messages to be lexically and semantically correct prior to verification via model checking. Hence behavior signatures deal with low-level functional implementation constraints, while USDL deals with higher-level real world concepts. However, USDL and behavioral signatures can be regarded as complementary concepts when taken in the context of real world service composition and both technologies are currently being used in the development of a commercial services integration tool [22].

8 Conclusions and Future Work

To catalogue, search and compose Web services in a semi-automatic to fully-automatic manner we need infrastructure to publish Web services, document them and query

repositories for matching services. Our semantics-based approach uses USDL to formally document the semantics of Web services and our discovery and composition engines find substitutable and composite services that best match the desired service.

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. Use of Constraint Logic Programming helped greatly in obtaining an efficient implementation of this system.

Our future work includes extending our engine to work with other web services description languages like OWL-S, WSMML, WSDL-S, etc. This should be possible as long as semantic relations between concepts are provided. It will involve extending the *TripleGenerator*, *QueryReader*, and *SemanticRelationsGenerator* modules. We would also like to extend our engine to support an external database to save off pre-processed data. This will be particularly useful when service repositories grow extremely large in size which can easily be the case in future. Future work also includes developing an industrial-strength system based on the research reported in this paper, in conjunction with a system that allows (semi-) automatic generation of USDL descriptions from the code and documentation of a service [22].

References

1. A. Bansal, S. Kona, L. Simon, A. Mallya, G. Gupta, and T. Hite. A Universal Service-Semantics Description Language. In *ECOWS*, pp. 214-225, 2005.
2. S. Kona, A. Bansal, L. Simon, A. Mallya, G. Gupta, and T. Hite. USDL: A Service-Semantics Description Language for Automatic Service Discovery and Composition. Tech. Report UTDCS-18-06. www.utdallas.edu/~sxxk038200/USDL.pdf.
3. S. McIlraith, T.C. Son, H. Zeng. Semantic Web Services. In *IEEE Intelligent Systems Vol. 16, Issue 2*, pp. 46-53, March 2001.
4. OWL-S www.daml.org/services/owl-s/1.0/owl-s.html.
5. WSMML: Web Service Modeling Language. www.wsmo.org/wsmml/.
6. WSDL-S: Web Service Semantics. <http://www.w3.org/Submission/WSDL-S>.
7. WSDL: Web Services Description Language. <http://www.w3.org/TR/wsdl>.
8. A. Bansal, K. Patel, G. Gupta, B. Raghavachari, E. Harris, and J. Staves. Towards Intelligent Services: A case study in chemical emergency response. In *ICWS*, pp.751-758, 2005.
9. OWL WordNet <http://taurus.unine.ch/knowler/wordnet.html>.
10. K. Marriott and P. Stuckey. *Prog. with Constraints: An Introduction*. MIT Press, 1998.
11. L. Sterling and S. Shapiro. *The Art of Prolog*. MIT Press, 1994.
12. OWL: Web Ontology Language Reference. <http://www.w3.org/TR/owl-ref>.
13. WS Challenge 2006. <http://insel.flp.cs.tu-berlin.de/wsc06>.
14. U. Keller, R. Lara, H. Lausen, A. Polleres, and D. Fensel. Automatic Location of Services. In *European Semantic Web Conference, May 2005*.
15. R. Hull and J. Su. Tools for design of composite Web services. In *SIGMOD*, 2004.
16. B. Srivastava. Automatic Web Services Composition using planning. In *KBCS*, pp.467-477.
17. S. McIlraith, S. Narayanan. Simulation, verification and automated composition of Web services. In *World Wide Web Conference, 2002*.
18. D. Mandell, S. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In *ISWC*, 2003.
19. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic Matching of Web Service Capabilities. In *ISWC*, pages 333-347, 2002.
20. S. Grimm, B. Motik, and C. Preist. Variance in e-Business Service Discovery. In *Semantic Web Services Workshop at ISWC, November 2004*.
21. M. Pistore, P. Roberti, and P. Traverso. Process-Level Composition of Executable Web Services. In *European Semantic Web Conference, pages 62-77, 2005*.
22. Metallec IQ Server. http://metallec.com/downloads/Metallec_Product_Brief_IQServer.pdf