Efficient Web Service Discovery and Composition using Constraint Logic Programming

Srividya Kona, Ajay Bansal, Gopal Gupta[1]
and Thomas D. Hite[2]

[1] Department of Computer Science
The University of Texas at Dallas
[2] Metallect 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. This automation can take place effectively only if formal semantic descriptions of Web services are available. In this paper we present an approach for automatic service discovery and composition with both syntactic and semantic description of Web services. In syntactic case, we use a repository of services described using WSDL (Web Service Description Language). In the semantic case, the services are described using USDL (Universal Service-Semantics Description Language), a language we have developed for formally describing the semantics of Web services. In this paper we show how the challenging task of building service discovery and composition engines can be easily implemented and efficiently solved via (Constraint) Logic programming techniques. We evaluate the algorithms on repositories of different sizes and show 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. The next 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 "programmatic 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]. Several efforts are underway to build such an infrastructure. 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, it only addresses the syntactical aspects of a Web service [17].

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 discovery and composition of Web services. We show how these tasks can be performed using both syntactic and semantic descriptions of Web services.

The rest of the paper is organized as follows. We present different approaches to the description of Web services in section 2 with brief description of WSDL and USDL. Section 3 describes the two major Web services tasks namely discovery and composition with their formal definitions. In section 4, we present our multi-step narrowing based solution for automatic service discovery and composition. Then we show the high-level design of our system with brief descriptions of the different components in section 5. Various efficiency and scalability issues are discussed in section 6. Then we show performance results of our discovery and composition algorithm in section 7. Finally we present our conclusions.

2 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. Currently WSDL (Web Services Description Language) [7] is used to describe Web services, but it is only syntactic in nature. The automation of Web service tasks (discovery, composition, etc.) can take place effectively only if formal semantic descriptions of Web services are available. For formally describing the semantics of Web services we have developed a language called USDL (Universal Service-Semantics Description Language). The motivation and details of USDL can be found in [1]. This section presents an overview of both the syntactic approach (WSDL) and the semantic approach (USDL) for description of Web services.

2.1 WSDL: Web Services Description Language

WSDL is an XML-based language used for describing the interface of a Web service. It describes services as a set of operations (grouped into ports) operating on messages containing either document-oriented or procedure-oriented information. WSDL descriptions are purely syntactic in nature, that is, they merely specify the format of messages to be exchanged by invocable operations.

Below is an example WSDL description of a *FlightReservation* service, similar to a service in the SAP ABAP Workbench Interface Repository for flight reservations [9], that takes in a *CustomerName*, *FlightNumber*, and *DepartureDate* as inputs and produces a *FlightReservation* as the output.

```
<definitions ...>
  <portType name="ReserveFlight_Service">
    <operation name="ReserveFlight">
      <input message="ReserveFlight_Request"/>
      <output message="ReserveFlight_Response"/>
    </operation>
  </portType>
  <message name="#ReserveFlight_Request">
    <part name="#CustomerName" type="xsd:string">
    <part name="#FlightNumber" type="xsd:string">
    <part name="#DepartureDate" type="xsd:date">
  </message>
  <message name="ReserveFlight_Response">
    <part name="FlightReservation" type="xsd:string"/>
  </message>
</definitions>
```

In order to allow interoperability and machine-readability of web documents, a common conceptual ground must be agreed upon. The first step towards this common ground are standard languages such as WSDL and OWL [15]. However, these do not go far enough, as for any given type of service there are numerous distinct representations in WSDL and for high-level concepts (e.g., a ternary predicate), there are numerous disparate representations in terms of OWL, representations that are distinct in terms of OWL's formal semantics, yet equal in the actual concepts they model. This is known as the semantic aliasing problem: distinct syntactic representations with distinct formal semantics yet equal conceptual semantics. For the semantics to equate things that are conceptually equal, we need to standardize a sufficiently comprehensive set of basic concepts, i.e., a universal ontology, along with a restricted set of connectives.

Industry specific ontologies along with OWL can also be used to formally describe Web services. This is the approach taken by the OWL-S language [4]. The problem with this approach is that it requires standardization and undue foresight. Standardization is a slow, bitter process, and industry specific ontologies would require this process to be iterated for each specific industry. Furthermore, reaching a industry specific standard ontology that is comprehensive and free of semantic aliasing is even more difficult. Undue foresight is required because many useful Web services will address innovative applications and industries that don't currently exist. Standardizing an ontology for travel and finances is easy, as these industries are well established, but new innovative services in new upcoming industries also need to be ascribed formal meaning. A universal ontology will have no difficulty in describing such new services.

2.2 USDL: Universal Service-Semantics Description Language

USDL is a language that service developers can use to specify formal semantics of Web services [1]. We need an ontology that is somewhat coarse-grained yet universal, and at a similar conceptual level to common real world concepts. WordNet [10] is a sufficiently comprehensive ontology that meets these criteria. USDL uses OWL WordNet ontology [11] thus providing a universal, complete, and tractable framework, which lacks the semantic aliasing problem, to which Web service messages and operations are mapped. As long as this mapping is precise and sufficiently expressive, reasoning can be done within the realm of OWL by using automated inference systems (such as, one based on description logic), and thus automatically reaping the wealth of semantic information in the OWL WordNet ontology that describes relations between ontological concepts, like subsumption (hyponym-hypernym) and equivalence (synonym) relations.

USDL can be regarded as providing semantics to WSDL statements. Thus, if WSDL can be regarded as a language for formally specifying the syntax of Web services, USDL can be regarded as a language for formally specifying their semantics. USDL allows sophisticated conceptual modeling and searching of available Web services, automated composition, and other forms of automated service integration. For example, the WSDL syntax and USDL semantics of Web services can be published in a directory which applications can access to automatically discover services. USDL is perhaps the first attempt to capture the semantics of Web services in a universal, yet decidable manner. Instead of documenting the function of a service as comments in English, one can write USDL statements that describe the function of that service. USDL relies on a universal ontology (OWL WordNet Ontology) to specify the semantics of atomic services.

USDL describes a service in terms of *portType* and *messages*, similar to WSDL. The formal class definitions and properties of USDL in OWL are available at [12]. The semantics of a service is given using the OWL WordNet ontology: portType (operations provided by the service) and messages (operation parameters) are mapped to disjunctions of conjunctions of (possibly negated) concepts in the OWL WordNet ontology. The semantics is given in terms of how a service *affects* the external world. USDL assumes that each side-effect is one of following four operations: *create, update, delete,* or *find*. A generic *affects* side-effect is used when none of the four apply. An application that wishes to use a service automatically should be able to reason with WordNet atoms using the OWL WordNet ontology. The syntactic terms describing portType and messages are mapped to disjunctions of conjunctions of (possibly negated) OWL WordNet ontological terms. A service is then formally defined as a function, labeled by the side-effect. Using USDL, conditions/constraints on the service can also be described. Below is the USDL description of the *FlightReservation* service.

```
<definitions>
  <portType rdf:about="#Flight_Reservation">
    <hasOperation rdf:resource="#ReserveFlight">
  </portType>
  <operation rdf:about="#ReserveFlight">
```

```
      <hasInput rdf:resource="#ReserveFlight_Request"/>
      <hasOutput rdf:resource="#ReserveFlight_Response"/>
      <creates rdf:resource="#FlightReservation" />
    </operation>
    <Message rdf:about="#ReserveFlight_Request">
      <hasPart rdf:resource="#CustomerName"/>
      <hasPart rdf:resource="#FlightNumber"/>
      <hasPart rdf:resource="#DepartureDate"/>
    </Message>
    <QualifiedConcept rdf:about="#CustomerName">
      <isA rdf:resource="#Name"/>
      <ofKind rdf:resource="#Customer"/>
    </QualifiedConcept>
    <BasicConcept rdf:about="#Name">
      <isA rdf:resource="&wn;name"/>
    </BasicConcept>
    <BasicConcept rdf:about="#Customer">
      <isA rdf:resource="&wn;customer"/>
    </BasicConcept>
    <!-- Similarly FlightNumber, DepartureDate are defined -->
  </definitions>
```

3 Automated Web service Discovery and Composition

Discovery and Composition are two of the major tasks related to Web services. In this section we formally describe these tasks as *The Discovery Problem* and *The Composition Problem*. Both these problems have a syntactic and a semantic version which are also described below.

3.1 The Discovery Problem

Given a repository of Web services, and a query (i.e., the requirements of the requested 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. This problem comes in two flavors: syntactic and semantic, depending on the type of service descriptions provided in the repository. All those services that produce at least the requested output parameters and use only from the provided input parameters can be valid solutions. Some of the solutions may be a little over-qualified, but they are still considered as long as they fulfill the input and output parameter requirements.

Definition: Let $\mathcal{R}$ be the set of services in a Web services repository. For simplicity, a service is represented as a pair of its input and output sets. Then let $\mathcal{Q} = (\mathcal{I}', \mathcal{O}')$ be a query service. The Discovery problem can be defined as automatically finding a set $\mathcal{S}$ of services such that $\mathcal{S} = \{s \mid s = (\mathcal{I}, \mathcal{O}),\ \mathrm{s} \in \mathcal{R}, \mathcal{I} \sqsubseteq \mathcal{I}',$

$\mathcal{O} \sqsupseteq \mathcal{O}'\}$. The meaning of the $\sqsubseteq$ relation depends on whether it is syntactic or semantic discovery. For syntactic discovery the $\sqsubseteq$ relation is the subset relation and for semantic discovery it is the subsumption (subsumes) relation. Figure 1 explains the discovery problem pictorially.

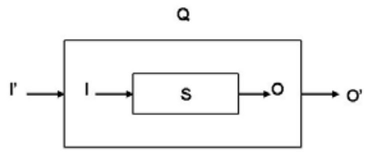


Fig. 1. Substitutable Service

Syntactic Discovery: WSDL provides syntactic description of Web services which can be provided in a repository. Given a query with requirements of the requested service, the discovery problem involves finding a specific service that can fulfill the given input and output criteria in the query based on a syntactical equivalence of the input and output names.

Semantic Discovery: We assume that a directory of services has already been compiled, and that this directory includes a USDL description document for each service. Inclusion of the USDL description, makes service 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.

3.2 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 chain of services that can be put together in correct order of execution to obtain the desired service. This problem also can be either syntactic or semantic depending on the kind of service descriptions provided in the repository. Web service discovery problem can be treated as a special case of the Web service composition problem where the length of the chain of services is one.

Definition: Let $\mathcal{R}$ be the set of services in a Web services repository. For simplicity, a service is represented as a pair of its input and output sets. Then let $\mathcal{Q} = (\mathcal{I}', \mathcal{O}')$ be a query service. The Composition problem can be defined as automatically finding a sequence $\mathcal{S}$ of services such that $\mathcal{S} = (\ \mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_n\ )$ where for all i, $\mathcal{S}_i \in \mathcal{R}$, $\mathcal{S}_i = (\mathcal{I}_i, \mathcal{O}_i)$ and $\mathcal{I}' \sqsupseteq \mathcal{I}_1$, $\mathcal{O}_1 \sqsupseteq \mathcal{I}_2$, ..., $\mathcal{O}_n \sqsupseteq \mathcal{O}'$.

The meaning of the $\sqsubseteq$ relation depends on whether it is syntactic or semantic composition. For syntactic composition the $\sqsubseteq$ relation is the subset relation and for semantic composition it is the subsumption (subsumes) relation. Figure 2 explains the composition problem pictorially.

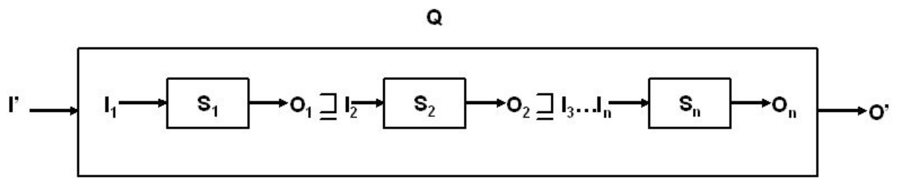


Fig. 2. Composite Service

Syntactic Composition: WSDL descriptions are provided in the repository. The Composition problem involves deriving a possible sequence of services where only the provided input parameters are used for the services and at least the required output parameter is provided as an output by the chained services. The goal is to derive a single solution, where the aim is to keep the list of involved services minimal. In the sequence of services, the outputs of a service are fed in as inputs of the next subsequent service.

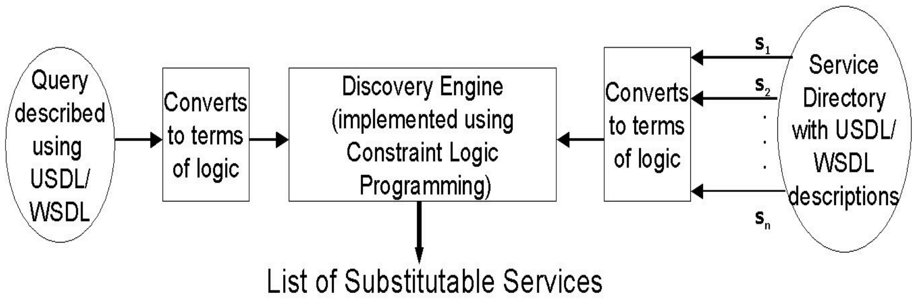
Semantic Composition: USDL descriptions are provided in the repository. For service composition, the first step is finding the set of composable services. USDL itself can be used to specify the requirements of the composed service that an application developer is seeking. Using the discovery engine, individual services that make up the composed service can be selected. Part substitution technique [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 $\mathcal{S}_1$ is composed with subservice $\mathcal{S}_2$, then the post-conditions of $\mathcal{S}_1$ must imply the pre-conditions of $\mathcal{S}_2$.

4 A Multi-step Narrowing based Solution

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.

4.1 Discovery Algorithm:

The Discovery routine takes in the query parameters and produces a list of matching services. Our algorithm first uses the query output parameters to narrow down the list of services in the repository. It gets all those services that produce at least the query outputs. In case of the semantic approach, the output parameters provided by a service must be equivalent to or be subsumed by the required output in the query. 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 wrong/bad inputs, say *WI* 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 *WI*. After all such services are removed, the remaining list is our final list of services called *Result*. Figure 3 shows a pictorial representation of our discovery engine.



Query described using USDL/ WSDL
Converts to terms of logic
Discovery Engine (implemented using Constraint Logic Programming)
Converts to terms of logic
s_1
s_2
s_n
Service Directory with USDL/ WSDL descriptions
List of Substitutable Services

Fig. 3. Discovery Engine

```
Algorithm: Discovery
Input: QI - QueryInputs, QO - QueryOutputs
Output: Result - ListOfServices
1. L ← NarrowServiceList(QO);
2. I ← GetAllInputParameters(L);
3. WI ← GetWrongInputs(I, QI); i.e., WI = I - QI
4. Result ← FilterServicesWithWrongInputs(WI, L);
5. Return Result;
```

4.2 Composition Algorithm:

The composition routine also starts with the query output parameters. It first finds a list of all those services which produce outputs such that they are equivalent to or are subsumed by the required output in the query. From the list obtained, for each service the algorithm fetches their input parameters, say I' and tries to find all those services from the repository that produce I' as outputs. The goal is to derive a single solution, which is a list of services that can be composed together to produce the requested service in the query. The aim is also to keep the list of involved services minimal. Figure 4 shows a pictorial representation of our composition engine.

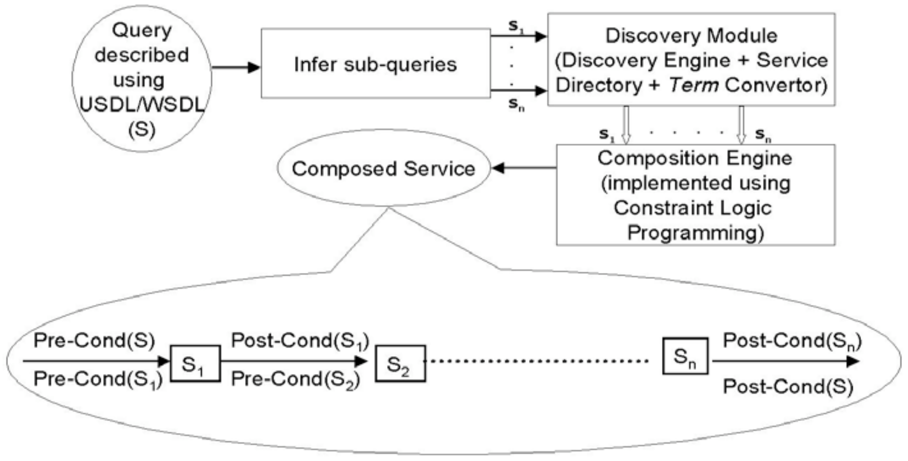

Query described using USDL/WSDL (S)
Infer sub-queries
s_1
s_n
Discovery Module (Discovery Engine + Service Directory + *Term* Convertor)
s_1
s_n
Composition Engine (implemented using Constraint Logic Programming)
Composed Service
Pre-Cond(S)
Pre-Cond(S_1)
S_1
Post-Cond(S_1)
Pre-Cond(S_2)
S_2
S_n
Post-Cond(S_n)
Post-Cond(S)

Fig. 4. Composition Engine

```
Algorithm: Composition
Input: QI - QueryInputs, QO - QueryOutputs
Output: Result - ListOfServices
1. L ← NarrowServiceList(QO);
2. For each service S in L
3.     Add S to the Result List;
4.     I ← GetAllInputParameters(S);
5.     L′ ← NarrowServiceList(I); i.e. find services which produce I as output
6.     Repeat the loop lines 2-5 on the new List L′;
7. End For
8. Return Result;
```

5 Implementation

Our discovery and composition engine is implemented using Prolog [14] with Constraint Logic Programming over finite domain [13], referred to as CLP(FD) hereafter. The high-level design of the Discovery and Composition engines is shown in Figure 5. The software system is made up of the following components.

5.1 Triple Generator

The triple generator module converts each service description into a triple. In syntactic approach WSDL descriptions are converted to triples like:

(null, affects(null, I, O), null).
WSDL being syntactic in nature, does not provide any information about Pre/Post-Conditions and side-effects. So we use the generic *affects* for all services. In the semantic approach the 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.

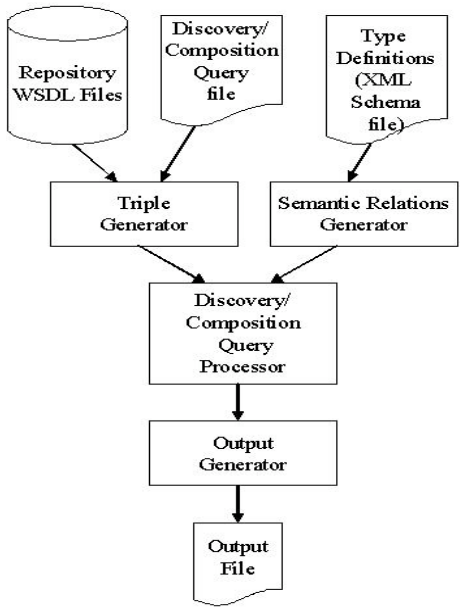

Repository WSDL Files
Discovery/ Composition Query file
Type Definitions (XML Schema file)
Triple Generator
Semantic Relations Generator
Discovery/ Composition Query Processor
Output Generator
Output File

Fig. 5. High-level Design

5.2 Query Reader

This module reads the query file and passes it on to the Triple Generator. The query file can be any pre-decided format. For example, the following XML snippet shows an example of a query file we use for testing our system.

```
<DiscoveryRoutine name="discovery1">
  <Provided>
    StreetAddress,CityAddress,StateAddress,ZipAddress
  </Provided>
  <Resultant>
    hotelName,hotelID
  </Resultant>
</DiscoveryRoutine>
```

In the above snippet the *Provided* tag holds the list of input requirements and the *Resultant* tag holds the list of output requirements.

5.3 Semantic Relations Generator

For the semantic approach, matching is done based on the semantic relations between the parameters, conditions/constraints if provided and side-effects if provided. 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.

5.4 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 parts 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 parts 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 the services, their input and output parameters and the semantic relations between the parameters. The following is the code snippet of our discovery 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).
```

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*.

5.5 Composition Query Processor

For service composition, the first step is finding the set of composable services. If a subservice $\mathcal{S}_1$ is composed with subservice $\mathcal{S}_2$, then the output parts of $\mathcal{S}_1$ must be the input parts of $\mathcal{S}_2$. Thus the processor has to find a set of services such that the outputs of the first service are inputs to the next service and so on. These services are then stitched together to produce the desired service.

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(Qname, A) :-
       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), decodeS(SA,A).

comp(_, Iset, A, A) :- empty_fdset(Iset),!.
comp(QIset, Iset, A, SA) :-
       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).
```

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 composition engine uses the built-in, higher order predicate 'bagof' to return all possible *ListOfServices* that can be composed to get the requested *queryService*.

5.6 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.

6 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 the Web service discovery and composition problem. It is because of these features, we decided on the Multi-step narrowing based approach to solving these problems and implemented it using Constraint Logic Programming.

Pre-processing: Our system initially pre-processes the repository and converts all service descriptions into Prolog terms. In case of semantic approach, 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 optimized 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 7. 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 grow 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 will be easily possible thus avoiding recomputation of the 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.

7 Performance

We evaluated our approach on different size repositories and tabulated the Preprocessing time and the Query Execution time. We noticed that there was a significant difference in the pre-processing time between the first and the 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 the subsequent runs. We used repositories from the WS-Challenge web site [16].

Table 1 shows performance results for our Discovery Algorithm and table 2 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 6 and 7 respectively. 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 service repository. However, what is surprising is the efficiency of service query processing, which is negligible (just 1 to 3 milliseconds) even for complex queries with large service repositories.

Repository Size (number of services)	*Number of I/O parameters*	*Non-Cached Pre-processing Time (in secs)*	*Cached Pre-processing Time (in secs)*	*Query Execution Time (in msecs)*
2000	4-8	36.5	7.3	1
2000	16-20	45.8	13.4	1
2000	32-36	57.8	23.3	2
2500	4-8	47.7	8.7	1
2500	16-20	58.7	16.6	1
2500	32-36	71.6	29.2	2
3000	4-8	56.8	12.1	1
3000	16-20	77.1	19.4	1
3000	32-36	88.2	33.7	3

Table 1. Performance of our Discovery Algorithm

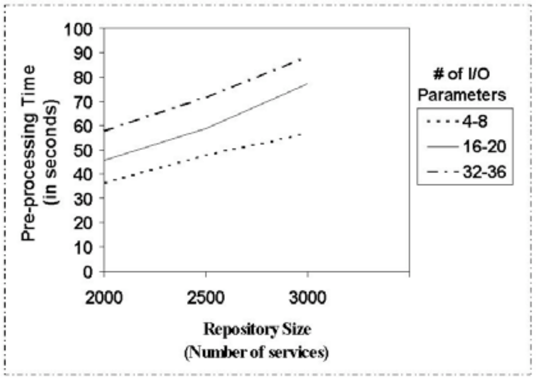


Fig. 6. Performance of Discovery Algorithm

Repository Size (number of services)	*Number of I/O parameters*	*Non-Cached Pre-processing Time (in secs)*	*Cached Pre-processing Time (in secs)*	*Query Execution Time (in msecs)*
2000	4-8	36.1	7.2	1
2000	16-20	47.1	15.1	1
2000	32-36	60.2	24.7	1
3000	4-8	58.4	11.0	1
3000	16-20	60.1	17.8	1
3000	32-36	102.0	42.1	1
4000	4-8	71.2	13.4	1
4000	16-20	87.9	25.3	1
4000	32-36	129.2	43.1	1

Table 2. Performance of our Composition Algorithm

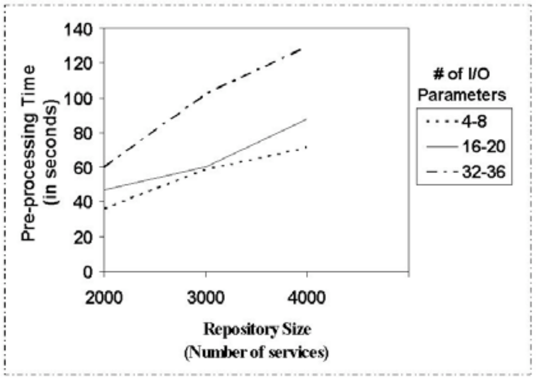


Fig. 7. Performance of Composition Algorithm

8 Conclusion

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 syntactic approach uses WSDL descriptions and applies the discovery and composition routines on first-order logic terms obtained from these descriptions. Our semantic 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.

Our solution produces accurate and quick results with both syntactic and semantic description of Web services. 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.

References

1. 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.
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/~sxk038200/USDL.pdf`.
3. S. McIlraith, T.C. Son, H. Zeng. Semantic Web Services. In IEEE Intelligent Systems Vol. 16, Issue 2, pp. 46-53, Mar. 2001.
4. OWL-S: Semantic markup for Web services. `www.daml.org/services/owl-s/1.0/owl-s.html`.
5. WSML: Web Service Modeling Language. `www.wsmo.org/wsml/`.
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. D. Harris, and J. C. Staves. Towards Intelligent Services: A case study in chemical emergency response. In *International Conference on Web Services*, pp. 751-758, 2005.
9. SAP Interface Repository. `http://ifr.sap.com/catalog/query.asp`.
10. WordNet: A Lexical Database for the English Language. `http://www.cogsci.princeton.edu/~wn`.
11. OWL WordNet: Ontology-based information management system. `http://taurus.unine.ch/knowler/wordnet.html`.
12. S. Kona, A. Bansal, G. Gupta, and T. Hite. USDL - Formal Definitions in OWL. Internal Report, University of Texas, Dallas, 2006. Available at `http://www.utdallas.edu/~srividya.kona/USDLFormalDefinitions.pdf`.
13. K. Marriott and P. J. Stuckey. Programming with Constraints: An Introduction. *MIT Press, 1998*.
14. L. Sterling and S. Shapiro. The Art of Prolog. *MIT Press, 1994*.
15. OWL: Web Ontology Language Reference. `http://www.w3.org/TR/owl-ref`.
16. WS Challenge 2006. `http://insel.flp.cs.tu-berlin.de/wsc06`.
17. U. Keller, R. Lara, H. Lausen, A. Polleres, and D. Fensel. Automatic Location of Services. In *European Semantic Web Conference*, May 2005.