Efficient Web Service Discovery and Composition using Constraint Logic Programming

Srividya Kona, Ajay Bansal, and Gopal Gupta
Department of Computer Science
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. 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 semantic description of Web services provided by 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 [16].

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 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 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. This section presents the current approaches used for describing Web services and their limitations. We then present an overview of the language that we developed called USDL (Universal Service-Semantics Description Language) which can formally describe the semantics of Web services. The motivation and details of USDL can be found in [1].

2.1 Current Approaches

Some of the popular approaches for describing semantics of Web services are OWL-S [4], WSML [5], and WSDL-S [6]. In this section we try to identify the similarities and differences of USDL with these approaches. OWL-S is a service description language which attempts to address the problem of semantic description via a highly detailed service ontology. But OWL-S also allows for complicated combining forms, which seem to defeat the tractability and practicality of OWL-S. The focus in the design of OWL-S is to describe the structure of a service in terms of how it combines other sub-services (if any used). The description of atomic services in OWL-S is left under-specified [19]. OWL-S includes the tags presents to describe the service profile, and the tag describedBy to describe the service model. The profile describes the (possibly conditional) states that exist before and after the service is executed. The service model describes how the service is (algorithmically) constructed from other simpler services. What the service actually accomplishes has to be inferred from these two descriptions in OWL-S. Given that OWL-S uses complicated combining forms, inferring the task that a service actually performs is, in general, undecidable. In contrast, in USDL, what the service actually does is directly described (via the verb affects and its refinements create, update, delete, and find).

OWL-S recommends that atomic services be defined using domain specific ontologies. Thus, OWL-S needs users describing the services and users using the services to know, understand and agree on domain specific ontologies in which the services are described. Hence, annotating services with OWL-S is a very time consuming, cumbersome, and invasive process. The complicated nature of OWL-S’s combining forms, especially conditions and control constructs, seems to allow for the semantic aliasing problem [19]. Other recent approaches such as WSMO, WSML, WSDL-S, etc., suffer from the same limitation [17]. In contrast, USDL uses the universal WordNet ontology to solve this problem.

Note that USDL and OWL-S can be used together. A USDL description can be placed under the describedBy tag for atomic processes, while OWL-S can be used to compose atomic USDL services. Thus, USDL along with WordNet can be treated as the universal ontology that can make an OWL-S description complete. USDL documents can be used to describe the semantics of atomic services that OWL-S assumes will be described by domain specific ontologies and pointed to by the OWL-S describedBy tag. In this respect, USDL and OWL-S are complementary: USDL can be treated as an extension to OWL-S which makes OWL-S description easy to write and semantically more complete.

OWL-S can also be regarded as the composition language for USDL. If a new service can be built by composing a few already existing services, then this new service can be described in OWL-S using the USDL descriptions of the existing services. Next, this new service can be automatically generated from its OWL-S description. The control constructs like Sequence and If-Then-Else of OWL-S allows us to achieve this. Note once a composite service has been defined using OWL-S that uses atomic services described in USDL, a new USDL description must be written for this composite service (automatic generation of this description is currently being investigated). This USDL description is the formal documentation of the new composite service and will make it automatically searchable once the new service is placed in the directory service. It also allows this composite service to be treated as an atomic service by some other application.

For example, the aforementioned ReserveFlight service which creates a flight reservation can be viewed as a composite process of first getting the flight details, then checking the flight availability and then booking the flight (creating the reservation). If we have these three atomic services namely GetFlightDetails, CheckFlightAvailability and BookFlight then we can create our ReserveFlight service by composing these three services in sequence using the OWL-S Sequence construct. The following is the OWL-S description of the composed ReserveFlight service.

```xml
<rdf:RDF xmlns:rdf ... >
  <process:CompositeProcess rdf:ID="ReserveFlight">
    <process:composedOf>
      <process:Sequence>
        <process:components rdf:parseType="Collection">
          <process:AtomicProcess rdf:about="#GetFlightDetails"/>
          <process:AtomicProcess rdf:about="#CheckFlightAvailability"/>
        </process:components>
      </process:Sequence>
    </process:composedOf>
  </process:CompositeProcess>
</rdf:RDF>
```

We can generate this composed ReserveFlight service
automatically. The component services can be discovered from existing services using their USDL descriptions. Once we have the component services, the OWL-S description can be used to generate the new composed service.

2.2 Motivation for USDL

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 [14]. 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.

We need an ontology that is somewhat coarse-grained yet universal, and at a similar conceptual level to common real world concepts. Currently there is only one sufficiently comprehensive ontology that meets these criteria: WordNet [9]. As stated, part of the common ground involves standardized languages such as OWL. For this reason, WordNet cannot be used directly, and instead we make use of an encoding of WordNet as an OWL base ontology [10]. Using an OWL WordNet ontology allows for our solution to use a universal, complete, and tractable framework, which lacks the semantic aliasing problem, to which we map Web service messages and operations. 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 we automatically reap the wealth of semantic information embodied in the OWL WordNet ontology that describes the relationships between ontological concepts, especially subsumption (hyponym) and equivalence (synonym) relationships.

2.3 Overview of USDL

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 [9] is a sufficiently comprehensive ontology that meets these criteria. USDL uses OWL WordNet ontology [10] 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 [11]. 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.

```xml
<definitions>
  <portType rdf:about="#ReserveFlight_Service">
    <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="#FlightNumber"/>
    <hasPart rdf:resource="#FirstName"/>
    <hasPart rdf:resource="#LastName"/>
    <hasPart rdf:resource="#StartDate"/>
    <hasPart rdf:resource="#ReturnDate"/>
    <hasPart rdf:resource="#StartAirport"/>
    <hasPart rdf:resource="#EndAirport"/>
  </Message>

  <Message rdf:about="#ReserveFlight_Response">
    <hasPart rdf:resource="#FlightReservation"/>
  </Message>

  <QualifiedConcept rdf:about="#FlightNumber">
    <isA rdf:resource="#Number"/>
    <ofKind rdf:resource="#Flight"/>
  </QualifiedConcept>

  <QualifiedConcept rdf:about="#FirstName">
    <isA rdf:resource="#Name"/>
    <ofKind rdf:resource="#First"/>
  </QualifiedConcept>

  <QualifiedConcept rdf:about="#StartDate">
    <isA rdf:resource="#Date"/>
    <ofKind rdf:resource="#Start"/>
    <hasCondition rdf:resource="#greaterThanToday"/>
  </QualifiedConcept>

  <!-- Similarly Last, Return, Airport are defined as Qualified Concepts-->
</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.

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. All those services that produce at least the requested output parameters and post-conditions and use only from the provided input parameters and pre-conditions 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 parameters and pre and post condition requirements.

**Definition:** Let \( R \) be the set of services in a Web services repository. For simplicity, a service is represented as a tuple of its pre-conditions, inputs, outputs and post-conditions. Then let \( Q = (CI', I', O', CO') \) be a query service. The Discovery problem can be defined as automatically finding a set \( S \) of services such that \( S = \{ s | s = (CI, I, O, CO), s \in R, CI' \supset CI, I \subseteq I', CO \Rightarrow CO', O \supseteq O' \} \). The meaning of the \( \supseteq \) is the subsumption (subsumes) relation. Figure 1 explains the discovery problem pictorially.

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. 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 $R$ be the set of services in a Web services repository. For simplicity, a service is represented as a tuple of its pre-conditions, inputs, outputs and post-conditions. Then let $Q = (CI', I', O', CO')$ be a query service. The Composition problem can be defined as automatically finding a sequence $S$ of services such that $S = (S_1, S_2, ..., S_n)$ where for all $i$, $S_i \in R$, $S_i = (CI_i, I_i, O_i, CO_i)$ and $I_i \supseteq I_1, O_i \supseteq I_2, ..., O_n \supseteq O', CI' \Rightarrow CI_1, CO_1 \Rightarrow CI_2, ..., CO_n \Rightarrow CO'$.

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 $S_1$ is composed with subservice $S_2$, then the post-conditions of $S_1$ must imply the pre-conditions of $S_2$.

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, i.e., 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.

Algorithm: Discovery
Input: $QI$ - QueryInputs, $QO$ - QueryOutputs
Output: Result - ListOfServices
1. \( L \leftarrow \text{NarrowServiceList}(QO) \);
2. \( I \leftarrow \text{GetAllInputParameters}(L) \);
3. \( WI \leftarrow \text{GetWrongInputs}(I, QI) \); i.e., \( WI = I - QI \)
4. Result \( \leftarrow \text{FilterServicesWithWrongInputs}(WI, L) \);
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.

Algorithm: Composition
Input: \( QI - \text{QueryInputs}, QO - \text{QueryOutputs} \)
Output: Result - ListOfServices
1. \( L \leftarrow \text{NarrowServiceList}(QO) \);
2. For each service \( S \) in \( L \)
3. Add \( S \) to the Result List;
4. \( I \leftarrow \text{GetAllInputParameters}(S) \);
5. \( L' \leftarrow \text{NarrowServiceList}(I) \); 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 [13] with Constraint Logic Programming over finite domain [12], 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. USDL descriptions are converted to triples like:

\((\text{Pre-Conditions, affect-type(affected-object, I, O), Post-Conditions})\).

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.
5.2 Query Reader

This module reads the query file and passes it on to the Triple Generator. USDL itself can be used as such a query language. A USDL description of the desired service can be written, which is read by the query reader and converted to a triple.

5.3 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.

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,0), encodeParam(0,OL),
  narrowO(OL,SL), fd_set(SL,Sset),
  fdset_member(S_INDEX,Sset),
  getExtInpList(I,ExtList),
  encodeParam(ExtList,IL),
  list_to_fdset(0,OLset),
  serv(S_INDEX,SI,:),
  list_to_fdset(0,OLset),
  fdset_subtract(OLset,0,ILset),
  comp(ILset,ILset,[S_INDEX],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 $S_1$ is composed with subservice $S_2$, then the output parts of $S_1$ must be the input parts of $S_2$. 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,0,0), encodeParam(0,OL),
  narrowO(OL,SL), fd_set(SL,Sset),
  fdset_member(S_INDEX,Sset),
  getExtInpList(I,ExtList),
  encodeParam(ExtList,IL),
  list_to_fdset(0,OLset),
  serv(S_INDEX,SI,:),
  list_to_fdset(0,OLset),
  fdset_subtract(OLset,0,ILset),
  comp(ILset,Iset,[S_INDEX],SA), decodeS(SA,A).
```

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 Pre-processing 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 [15].

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.

8 Related Work

Discovery and composition of Web services has been active area of research recently [21, 18, 23, 24, 22]. 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. The approaches proposed by others also rely on a domain specific ontology (specified on OWL/DAML), and thus suffer from the problem mentioned earlier, namely, to discover/compose such services the discovery/composition engine has to be aware of the domain specific ontology. Thus, 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, hypernym, hyponym, meronym, and holonym relations are defined between the various terms of the domain specific ontology.

Another related area of research involves message conversation constraints, also known as behavioral signatures [20]. 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 tech-
nologies are currently being used in the development of a
commercial services integration tool.

9 Conclusion

To catalogue, search and compose Web services in a
semi-automatic to fully-automatic manner we need infras-
tructure 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 on large repositories.
Use of Constraint Logic Programming helped greatly
in obtaining an efficient implementation of this system.

References

and T. Hite. A Universal Service-Semantics Description
Language. In European Conference On Web Ser-

and T. Hite. USDL - A Service-Semantics Description
Language for Automatic Service Discovery and
udallas.edu/~sxk038200/USDL.pdf.

vices. In IEEE Intelligent Systems Vol. 16, Issue 2,

www.daml.org/services/owl-s/1.0/
owl-s.html.

wsml.org/.

w3.org/Submission/WSDL-S.

www.w3.org/TR/wSDL.

Harris, and J. C. Staves. Towards Intelligent Services:
A case study in chemical emergency response. In In-
ternational Conference on Web Services, pp. 751-758,
2005.

edu/~wn.

[10] OWL WordNet: Ontology-based information man-

USDL - Formal Definitions in OWL. Internal Re-
port, University of Texas, Dallas, 2006. Available
at http://www.utdallas.edu/~srividya.
kona/USDLFormalDefinitions.pdf.

[12] K. Marriott and P. J. Stuckey. Programming with Con-


/www.w3.org/TR/owl-ref.

tu-berlin.de/wsc06.

[16] U. Keller, R. Lara, H. Lausen, A. Polleres, and
D. Fensel. Automatic Location of Services. In Eu-

[17] A conceptual comparison between WSMO and OWL-

[18] S. McIlraith, T.C. Son. Adapting golog for composi-
tion of semantic Web services. In KRR, pages 482–
493, 2002.


[21] B. Srivastava, J. Koehler. Web Services Composition -
Current Solutions and Open Problems. In ICAPS,
2003.

[22] B. Srivastava. Automatic Web Services Composition
using planning. In KBCS, pages 467–477.

[23] S. McIlraith, S. Narayanan Simulation, verification
and automated composition of Web services. In World