Web Service Discovery and Composition using USDL

Srividya Kona, Ajay Bansal, 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

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. In this paper, we present the design of an automatic service discovery and composition engine using USDL (Universal Service-Semantics Description Language) [1, 2], a language for formally describing the semantics of web-services. The implementation will be used for the WS-Challenge 2006 [3].

1. Introduction

A web-service is a program available on a website that "effects some action or change" in the world (i.e., causes a side-effect). 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 themselves rather than by humans. To achieve this, 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 and composition.

In this paper we present the design of a software system for automatic service discovery and composition. This system uses web service descriptions written in USDL [1, 2]. In section 2 we present a brief overview of USDL. Section 3 shows the design of our software with brief descriptions of the different components of the system followed by conclusions and references.

2. Overview of USDL

USDL is a language that provides formal semantics of web-services thus allowing sophisticated conceptual modeling and searching of available services, automated composition, and automated service integration. This is an ongoing project with Metallect Corp. The design and formal specification in OWL was published in European Conference On Web Services, 2005 [1]. The WSDL [4] syntax and USDL semantics of web services can be published in a directory which applications can access to discover services. To provide formal semantics, a common denominator must be agreed upon that everybody can use as a basis of understanding the meaning of services. Additionally, the semantics should be given at a conceptual level that captures common real world concepts. USDL uses an ontology based on OWL WordNet [5] for a universal ontology of basic concepts upon which arbitrary meets and joins can be added in order to gain tractable flexibility.

USDL describes a service in terms of portType and messages, similar to WSDL. 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. Additional semantics is given in terms of how a service *affects* the external world. Formal semantic description of any external conditions (pre-conditions or post-conditions) acting on the service can also be provided using USDL.

3. Design

Our discovery and composition engine is written using Prolog [6]. Our software system for the WS-Challenge [3] comprises of the following components shown in Figure 1.

3.1. USDL Generator

This module parses all WSDL descriptions from the given repository and converts them into corresponding USDL descriptions. For the WS-Challenge, this module maps part names of a service to their corresponding type definitions from the XML Schema file, as opposed

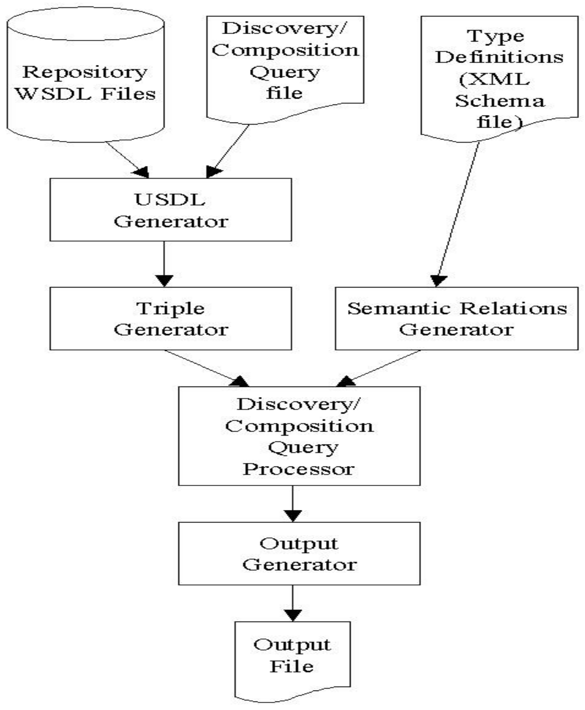
Repository WSDL Files
Discovery/ Composition Query file
Type Definitions (XML Schema file)
USDL Generator
Triple Generator
Semantic Relations Generator
Discovery/ Composition Query Processor
Output Generator
Output File

Figure 1. High-level Design

to pointing them to concepts in OWL WordNet ontology. Our approach does not need a separate query language. It parses the query file and converts each query for the desired service also into a USDL description.

3.2. Triple Generator

The triple generator module converts each service description into a triple as follows:

(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 of the service. 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 [7]. For the WS-Challenge, conditions on a service will not be provided and hence the *Pre-Conditions* and *Post-Conditions* in the triple will be null. The affect-type will also not be available and so this module will assign a generic affect to all services.

3.3. Semantic Relations Generator

For the semantic part of the challenge, we have to match the parmater types. The XML Schema will provide type hierarchy. We will obtain the semantic relations from the XML Schema file provided instead of OWL WordNet ontology. Complex types will be described in the schema file using simple data types, thus providing a type hierarchy. A supertype will subsume its subtype which is nothing but the hyponym and hypernym relation. A hyponym is a word that is more specific than a given word, also called the sub-ordinate. A hypernym is a word that is more generic than a given word, also called the super-ordinate.

This module will extract the type definitions from the XML Schema file and create the semantic relations (hypernym, hyponym, etc.) [7] between the different types in a format that the discovery and composition query processors will understand. For example, if the XML Schema has a type *p66a9128258* which is a supertype of *p56a4809967*, then this module will add the prolog fact

hyponym(p56a4809967,p66a9128258).

to the list of facts.

3.4. Discovery Query Processor

This module compares the discovery query with all the services in the repository. It uses an extended/special unification algorithm to find a matching term. The unification mechanism is different depending on the type of match (*Exact, Specific, Generic, Part* or *Whole*) required. The processor works as follows:

1. 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 type with hypernym relation [7], i.e., a *generic* substitutable.
2. 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 type with hyponym relation [7], i.e., a *specific* substitutable.

The discovery engine is written using prolog (a logic programming language) [6]. It uses a repository of facts, which contains a list of all the services and the semantic relations. The query is converted into a prolog query that looks as follows:

discoverServices(queryService, solutionService).

The engine will try to find a list of *solutionServices* that match the *queryService*. Our code for the engine will have various rules to solve the *discoverServices* query.

3.5. 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.

If the complete semantics of a web service are described using USDL and OWL WordNet ontology, *Part substitution* technique [7] 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. For the challenge, we do not have with mappings to OWL WordNet Ontology and description of conditions. So we will be using only *Exact, Generic, and Specific substitution* [7] techniques to find the list of composable services. Similar to the discovery engine, composition engine is also written using prolog. The query is converted into a prolog query that looks as follows:

composeServices(queryService, listOfServices).

The engine will try to find a *listOfServices* that can be composed into the requested *queryService*. Our code for the engine will have various rules to solve the *composeServices* query. Our prolog code will be setup in such a way that all possible *listofServices* that can be used for composition will be returned.

3.6. 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 using the specified XML format for the WS-Challenge[3].

4. Conclusion

To catalogue, search and compose services in a semi-automatic to fully-automatic manner we need infrastructure to publish services, document services and query repositories for matching services. Our approach uses USDL to formally document the semantics of services and our discovery and composition engines find substitutable services that best match the desired service.

Our solution will produce very good results when semantic descriptions of web services are provided using USDL. While matching real-world services, our discovery and composition engine will look at OWL WordNet ontology for the meanings. For the challenge, type hierarchy is the only semantics provided and hence we will not be able to use all the semantic relations available in WordNet. We apply some optimization techniques to our system so that it is efficient on the WS-Challenge repositories. We use constraint-logic programming [8] for better pruning of the search space. We have put in efforts for testing the efficiency of our system and identifying the correct optimization strategies.

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, 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`.

[3] WS Challenge 2006. `http://insel.flp.cs.tu-berlin.de/wsc06`.

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

[5] Ontology-based information management system, wordnet OWL-Ontology. `http://taurus.unine.ch/knowler/wordnet.html`.

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

[7] 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. Technical Report UTDCS-18-06, University of Texas, Dallas, 2006. Submitted to JWSR. Available at `http://www.utdallas.edu/~srividya.kona/USDL.pdf`.

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