USDL: A Service-Semantics Description Language for Automatic Service Discovery and Composition.*

Srividya Kona, Ajay Bansal, Luke Simon,
Ajay Mallya, and Gopal Gupta
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. This automation can take place only if a formal description of the web-services is available. In this paper we present an infrastructure using USDL (Universal Service-Semantics Description Language), a language for formally describing the semantics of web-services. USDL is based on the Web Ontology Language (OWL) and employs WordNet as a common basis for understanding the meaning of services. USDL can be regarded as formal service documentation that will allow sophisticated conceptual modeling and searching of available web-services, automated service composition, and other forms of automated service integration. A theory of service substitution using USDL is presented. The rationale behind the design of USDL along with its formal specification in OWL is presented with examples. We also compare USDL with other approaches like OWL-S, WSDL-S, and WSML and show that USDL is complementary to these approaches.

1 Introduction

A web-service is a program available on a web-site that "effects 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 themselves rather than by humans. 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 needs to be supported 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. Several efforts are underway to build such an infrastructure. These efforts include approaches based on the semantic web (such as OWL-S [5]) as well as those based on XML, such as Web Services Description Language (WSDL [7]). Approaches such

*This is an expanded version of the paper 'A Universal Service-Semantics Description Language' that appeared in European Conference On Web Services, 2005 [15] and received its best paper award.

as WSDL are purely syntactic in nature, that is, they merely specify the format of the service. In this paper we present an approach that is based on semantics. Our approach can be regarded as providing semantics to WSDL statements. We present the design of a language called Universal Service-Semantics Description Language (USDL) which service developers can use to specify formal semantics of web-services [14, 15]. 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 can be thought of as *formal service documentation* that will allow 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. That is, given a formal description of the context in which a service is needed, the service(s) that will precisely fulfill that need can be determined. The directory can then be searched for the exact service, or two or more services that can be composed to synthesize the required service, etc.

To provide formal semantics, a common denominator must be agreed upon that everybody can use as a basis of understanding the meaning of services. This common conceptual ground must also be somewhat coarse-grained so as to be tractable for use by both engineers and computers. That is, semantics of services should not be given in terms of low-level concepts such as Turing machines, first-order logic and their variants, since service description, discovery, and synthesis then become tasks that are practically intractable and theoretically undecidable. Additionally, the semantics should be given at a conceptual level that captures common real world concepts. Furthermore, it is too impractical to expect disparate companies to standardize on application (or domain) specific ontologies to formally define semantics of web-services, and instead a common universal ontology must be agreed upon with additional constructors. Also, application specific ontologies will be an impediment to automatic discovery of services since the application developer will have to be aware of the specific ontology that has been used to describe the semantics of the service in order to frame the query that will search for the service. The danger is that the service may not be defined using the particular domain specific ontology that the application developer uses to frame the query, however, it may be defined using some other domain specific ontology, and so the application developer will be prevented from discovering the service even though it exists. These reasons make an ontology based on OWL WordNet [2, 8] a suitable candidate for a universal ontology of basic concepts upon which arbitrary meets and joins can be added in order to gain tractable flexibility.

We describe the meaning of conceptual modeling and how it could be obtained via a common universal ontology based on WordNet in the next section. Section 3, gives a brief overview of how USDL attempts to semantically describe web-services. In section 4, we discuss precisely how a WSDL document can be prescribed meaning in terms of WordNet ontology. Section 5 gives a complete USDL annotation for a Hotel-Reservation service. In section 6 we present the theoretical foundations of service description and substitution in USDL. Automatic discovery of web-services using USDL is discussed in section 7. Composition of web-services using USDL is discussed in section 8. Comparison of USDL with other approaches like OWL-S and WSML is discussed in section 9. Section 10 shows related work. Finally, conclusions and future work are addressed in the last section.

2 A Universal Ontology

To describe service semantics, we should agree on a common ground to model our concepts. We can describe what any given web-service does from first principles using approaches based on logic. This is the approach taken by frameworks such as dependent type systems and programming logics prevalent in the field of software verification where a "formal understanding" of the service/software is needed in order to verify it. However, such solutions are both low-level, tedious, and undecidable to be of practical use. Instead, we are interested in modeling higher-level concepts. That is, we are more interested in answering questions such as, what does a service do from the end user's or service integrator's perspective, as opposed to the far more difficult questions, such as, what does the service do from a computational view? We care more about real world concepts such as "customer", "bank account", and "flight itinerary" as opposed to the data structures and algorithms used by a service to model these concepts. The distinction is subtle, but is a distinction of granularity as well as a distinction of scope.

In order to allow interoperability and machine-readability of our documents, a common conceptual ground must be agreed upon. The first step towards this common ground are standard languages such as WSDL and OWL. 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 [5]. 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 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. WordNet [8] is a sufficiently comprehensive ontology that meets these criteria. 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 [2]. Using an OWL WordNet ontology allows 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 relationships between ontological concepts, especially subsumption (hyponym-hypernym) and equivalence (synonym) relationships.

3 USDL: An Overview

As mentioned earlier, USDL can be regarded as a language to formally specify the semantics of web-services. It is perhaps the first attempt to capture the semantics of web-services in a universal, yet decidable manner. It is quite distinct from previous approaches such as WSDL and OWL-S [5]. As mentioned earlier, WSDL only defines syntax of the service; USDL provides the missing semantic component. USDL can be thought of as a formal language for service documentation. Thus, 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 is quite distinct from OWL-S, which is designed for a similar purpose, and as we shall see the two are in fact complementary. OWL-S primarily describes the states that exist before and after the service and how a service is composed of other smaller sub-services (if any). Description of atomic services is left under-specified in OWL-S. They have to be specified using domain specific ontologies; in contrast, atomic services are completely specified in USDL, and USDL relies on a universal ontology (OWL WordNet Ontology) to specify the semantics of atomic services. USDL and OWL-S are complementary in that OWL-S's strength lies in describing the structure of composite services, i.e., how various atomic services are algorithmically combined to produce a new service, while USDL is good for fully describing atomic services. Thus, OWL-S can be used for describing the structure of composite services that combine atomic services described using USDL.

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. The semantics is given in terms of how a service *affects* the external world. The present design of 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 make use of a service automatically should be able to reason with WordNet atoms using the OWL WordNet ontology.

We also define the formal semantics of USDL. As stated earlier, 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. The main contribution of our work is the design of a universal service-semantics description language (USDL), along with its formal semantics, and a theory of service substitution using it.

4 Design of USDL

The design of USDL rests on two formal languages: Web Services Description Language (WSDL) [7] and Web Ontology Language (OWL) [6]. The Web Services Description Language (WSDL) [7] is used to give a syntactic description of the name and parameters of a service. The description is syntactic in the sense that it describes the formatting of services on a syntactic level of method signatures, but is incapable of describing what concepts are involved in a service and what a service actually does, i.e., the conceptual semantics of the service. Likewise, the Web Ontology Language (OWL) [6], was developed as an extension to the Resource Description Framework (RDF) [3], both standards are designed to allow formal conceptual modeling via logical ontologies, and these languages also allow for the markup of existing web resources with semantic information from the conceptual models. USDL employs WSDL and OWL in order to describe the syntax and semantics

of web-services. WSDL is used to describe message formats, types, and method prototypes, while a specialized universal OWL ontology is used to formally describe what these messages and methods mean, on a conceptual level.

USDL can be regarded as the semantic counterpart to the syntactic WSDL description. WSDL documents contain two main constructs to which we want to ascribe conceptual meaning: messages and portType. These constructs are aggregates of service components which will be directly ascribed meaning. Messages consist of typed parts and portType consists of operations parameterized on messages. USDL defines OWL surrogates or proxies of these constructs in the form of classes, which have properties with values in the OWL WordNet ontology.

4.1 Concept

USDL defines a generic class called *Concept* which is used to define the semantics of parts of messages.

```
<owl:Class rdf:ID="Concept">
  <rdfs:comment>Generic class of USDL Concept</rdfs:comment>
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#BasicConcept"/>
    <owl:Class rdf:about="#QualifiedConcept"/>
    <owl:Class rdf:about="#InvertedConcept"/>
    <owl:Class rdf:about="#ConjunctiveConcept"/>
    <owl:Class rdf:about="#DisjunctiveConcept"/>
  </owl:unionOf>
</owl:Class>
```

The USDL *Concept* class denotes the conceptual objects constructed from the OWL WordNet ontology. For most purposes, message parts and other WSDL constructs will be mapped to a subclass of USDL *Concept* so that useful concepts can be modeled as set theoretic formulas of union, intersection, and negation of basic concepts. These subclasses of *Concept* are *BasicConcept*, *QualifiedConcept*, *InvertedConcept*, *ConjunctiveConcept*, and *DisjunctiveConcept*.

4.1.1 Basic Concept

An *BasicConcept* is the actual contact point between USDL and WordNet. This class acts as proxy for WordNet lexical entities.

```
<owl:Class rdf:about="#BasicConcept">
  <rdfs:subClassOf rdf:resource="#Concept"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isA"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
         1
      </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

The property cardinality restrictions require all USDL *BasicConcept*s to have exactly one defining value for the *isA* property. An instance of *BasicConcept* is considered to be equated with a WordNet lexical concept given by the *isA* property.

```
<owl:ObjectProperty rdf:ID="isA">
  <rdfs:domain rdf:resource="#BasicConcept"/>
  <rdfs:range rdf:resource="&wn;LexicalConcept"/>
</owl:ObjectProperty>
```

4.1.2 Qualified Concept

A *QualifiedConcept* is a concept classified by another lexical concept.

```
<owl:Class rdf:about="#QualifiedConcept">
  <rdfs:subClassOf rdf:resource="#Concept"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isA"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
         1
      </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>

  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#ofKind"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
         1
      </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

The property cardinality restrictions require all USDL *QualifiedConcept*s to have exactly one defining value for the *isA* property, and exactly one value for the *ofKind* property. An instance of *QualifiedConcept* is considered to be equated with a lexical concept given by the *isA* property and classified by a lexical concept given by the optional *ofKind* property.

```
<owl:ObjectProperty rdf:ID="isA">
  <rdfs:domain rdf:resource="#QualifiedConcept"/>
  <rdfs:range rdf:resource="#Concept"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="ofKind">
  <rdfs:domain rdf:resource="#QualifiedConcept"/>
  <rdfs:range rdf:resource="#Concept"/>
</owl:ObjectProperty>
```

4.1.3 Inverted Concept

In case of *InvertedConcept* the corresponding semantics are the complement of USDL concepts.

```
<owl:Class rdf:about="#InvertedConcept">
  <rdfs:subClassOf rdf:resource="#Concept"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasConcept"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
         1
```

```
      </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty rdf:ID="hasConcept">
  <rdfs:domain rdf:resource="#Concept"/>
  <rdfs:range rdf:resource="#Concept"/>
</owl:ObjectProperty>
```

4.1.4 Conjunctive and Disjunctive Concept

The *ConjunctiveConcept* and *DisjunctiveConcept* respectively denote the intersection and union of USDL *Concept*s.

```
<owl:Class rdf:about="#ConjunctiveConcept">
  <rdfs:subClassOf rdf:resource="#Concept"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasConcept"/>
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
         2
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="#DisjunctiveConcept">
  <rdfs:subClassOf rdf:resource="#Concept"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasConcept"/>
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
         2
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

The property cardinality restrictions on *ConjunctiveConcept* and *DisjunctiveConcept* allow for n-ary intersections and unions (where $n \geq 2$) of USDL concepts. For generality, these concepts are either *BasicConcepts*, *QualifiedConcepts*, *ConjunctiveConcepts*, *DisjunctiveConcepts*, or *InvertedConcepts*.

4.2 Affects

The *affects* property is specialized into four types of actions common to enterprise services: *creates*, *updates*, *deletes*, and *finds*.

```
<owl:ObjectProperty rdf:ID="affects">
  <rdfs:comment>
    Generic class of USDL Affects
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Operation"/>
  <rdfs:range rdf:resource="#Concept"/>
```

```
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#creates">
  <rdfs:subPropertyOf rdf:resource="#affects"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#updates">
  <rdfs:subPropertyOf rdf:resource="#affects"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#deletes">
  <rdfs:subPropertyOf rdf:resource="#affects"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#finds">
  <rdfs:subPropertyOf rdf:resource="#affects"/>
</owl:ObjectProperty>
```

Note that each of these specializations inherits the domain and range of the *affects* property. Most services can be described using one of these types of effects. For those services that cannot be described in terms of these specializations, the parent *affects* property can be used instead which is described as an USDL concept.

4.3 Conditions and Constraints

Services may have some external conditions (pre-conditions and post-conditions) specified on the input or output parameters. *Condition* class is used to describe all such constraints. Conditions are represented as conjunction or disjunction of binary predicates. Predicate is a trait or aspect of the resource being described.

```
<owl:Class rdf:ID="Condition">
  <rdfs:comment>
    Generic class of USDL Condition
  </rdfs:comment>
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#AtomicCondition"/>
    <owl:Class rdf:about="#ConjunctiveCondition"/>
    <owl:Class rdf:about="#DisjunctiveCondition"/>
  </owl:unionOf>
</owl:Class>

<owl:Class rdf:about="#AtomicCondition">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasConcept"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
         1
      </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>

  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#onPart"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
```

```
        1
      </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>

  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasValue"/>
      <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">
        1
      </owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

A condition has exactly one value for the *onPart* property and at most one value for the *hasValue* property, each of which is of type USDL *Concept*.

```
<owl:ObjectProperty rdf:ID="onPart">
  <rdfs:domain rdf:resource="#AtomicCondition"/>
  <rdfs:range rdf:resource="#Concept"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasValue">
  <rdfs:domain rdf:resource="#AtomicCondition"/>
  <rdfs:range rdf:resource="#Concept"/>
</owl:ObjectProperty>
```

4.3.1 Conjunctive and Disjunctive Conditions

The *ConjunctiveCondition* and *DisjunctiveCondition* respectively denote the conjunction and disjunction of USDL *Condition*s.

```
<owl:Class rdf:about="#ConjunctiveCondition">
  <rdfs:subClassOf rdf:resource="#Condition"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasCondition"/>
      <owl:minCardinality rdf:datatype= "&xsd;nonNegativeInteger">
         2
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="#DisjunctiveCondition">
  <rdfs:subClassOf rdf:resource="#Condition"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasCondition"/>
      <owl:minCardinality rdf:datatype= "&xsd;nonNegativeInteger">
         2
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

```
<owl:ObjectProperty rdf:ID="hasCondition">
  <rdfs:domain rdf:resource="#Concept"/>
  <rdfs:range rdf:resource="#Condition"/>
</owl:ObjectProperty>
```

The property cardinality restrictions on *ConjunctiveCondition* and *DisjunctiveCondition* allow for n-ary conjunctions and disjunctions (where $n \geq 2$) of USDL conditions. In general any n-ary condition can be written as a combination of conjunctions and disjunctions of binary conditions.

4.4 Messages

Services communicate by exchanging messages. As mentioned, messages are simple tuples of actual data, called parts. Take for example, a flight reservation service similar to the SAP ABAP Workbench Interface Repository for flight reservations [4], which makes use of the following message.

```
<message name="#ReserveFlight_Request">
  <part name="#CustomerName" type="xsd:string">
  <part name="#FlightNumber" type="xsd:string">
  <part name="#DepartureDate" type="xsd:date">
  ...
</message>
```

The USDL surrogate for a WSDL message is the *Message* class, which is a composite entity with zero or more parts. Note that for generality, messages are allowed to contain zero parts.

```
<owl:Class rdf:ID="Message">
  <rdfs:comment>
    Generic class of USDL Message
  </rdfs:comment>
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Input"/>
    <owl:Class rdf:about="#Output"/>
  </owl:unionOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasPart"/>
      <owl:minCardinality
       rdf:datatype="&xsd;nonNegativeInteger">
         0
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="#Input">
  <rdfs:subClassOf rdf:resource="#Message"/>
</owl:Class>

<owl:Class rdf:about="#Output">
  <rdfs:subClassOf rdf:resource="#Message"/>
</owl:Class>
```

Each part of a message is simply a USDL *Concept*, as defined by the *hasPart* property. Semantically messages are treated as tuples of concepts.

```
<owl:ObjectProperty rdf:ID="hasPart">
  <rdfs:domain rdf:resource="#Message"/>
  <rdfs:range rdf:resource="#Concept"/>
</owl:ObjectProperty>
```

Continuing our example flight reservation service, the *ReserveFlightRequest* message is given semantics using USDL as follows, where `&wn;customer` and `&wn;name` are valid XML references to WordNet lexical concepts.

```
<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 concepts FlightNumber and DepartureDate are defined -->
```

4.5 PortType

A service consists of portType, which is a collection of procedures or operations that are parametric on messages. Our example flight reservation service might contain a *portType* definition for a flight reservation service that takes as input an itinerary and outputs a reservation receipt.

```
<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>
```

The USDL surrogate is defined as the class *portType* which contains zero or more *Operation*`s` as values of the *hasOperation* property.

```
<owl:Class rdf:about="#portType">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource=#hasOperation"/>
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
         0
      </owl:minCardinality>
```

```
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty rdf:ID="hasOperation">
  <rdfs:domain rdf:resource="#portType"/>
  <rdfs:range rdf:resource="#Operation"/>
</owl:ObjectProperty>
```

As with the case of messages, portTypes are not directly assigned meaning via the OWL WordNet ontology. Instead the individual *Operation*s are described by their side-effects via an *affects* property. Note that the parameters of an operation are already given meaning by ascribing meaning to the messages that constitute the parameters.

```
<owl:Class rdf:about="#Operation">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasInput"/>
      <owl:minCardinality
       rdf:datatype="&xsd;nonNegativeInteger">
         0
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
       <owl:onProperty rdf:resource="#hasOutput"/>
       <owl:minCardinality
        rdf:datatype="&xsd;nonNegativeInteger">
          0
       </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#affects"/>
      <owl:minCardinality
       rdf:datatype="&xsd;nonNegativeInteger">
         1
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

An operation can have one or more values for the *affects* property, all of which are of type USDL *Concept*, which is the target of the effect.

```
<owl:ObjectProperty rdf:ID="hasInput">
  <rdfs:domain rdf:resource="#Operation"/>
  <rdfs:range rdf:resource="#Input"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasOutput">
  <rdfs:domain rdf:resource="#Operation"/>
  <rdfs:range rdf:resource="#Ouput"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="affects">
  <rdfs:domain rdf:resource="#Operation"/>
  <rdfs:range rdf:resource="#Concept"/>
</owl:ObjectProperty>
```

5 Semantic Description of a Service

This section shows an example syntactic description of a web-service using WSDL and its corresponding semantic description using USDL.

Hotel Reservation Service: The service described here is a simplified hotel-reservation service published in a web-service registry. This service can be treated as atomic: i.e., no interactions between buying and selling agents are required, apart from invocation of the service and receipt of its outputs by the buyer. Given certain inputs and pre-conditions, the service provides certain outputs and has specific effects.

This service takes in a *HotelChain*, *StartDate*, *NumNights*, *NumPersons*, *NumRooms*, *FirstName*, and *LastName* as input parameters. It has a few input pre-conditions that *NumNights*, *NumRooms* must be greater than zero and *StartDate* must be greater than today. This service outputs a *Reservation* at the end of transaction.

5.1 WSDL definition

The following is WSDL definition of the service. This service provides a single operation called *ReserveHotel*. The input and output messages are defined below. The conditions on the service cannot be described using WSDL.

```
<definitions ...>
  <portType name="ReserveHotel_Service">
    <operation name="ReserveHotel">
      <input message="ReserveHotel_Request"/>
      <output message="ReserveHotel_Response"/>
    </operation>
  </portType>

  <message name="ReserveHotel_Request">
    <part name="HotelChain" type="xsd:string"/>
    <part name="StartDate" type="xsd:date"/>
    <part name="NumNights" type="xsd:integer"/>
    <part name="NumPersons" type="xsd:integer"/>
    <part name="NumRooms" type="xsd:integer"/>
    <part name="FirstName" type="xsd:integer"/>
    <part name="LastName" type="xsd:integer"/>
  </message>

  <message name="ReserveHotel_Response">
    <part name="Reservation" type="xsd:string"/>
  </message>
  ...
</definitions>
```

5.2 USDL annotation

The following is the complete USDL annotation corresponding to the above mentioned WSDL description. The input pre-condition and the global constraint on the service are also described semantically.

```
<definitions>
  <portType rdf:about=
            "#ReserveHotel_Service">
    <hasOperation rdf:resource=
                  "#ReserveHotel"/>
  </portType>

  <operation rdf:about="#ReserveHotel">
    <hasInput  rdf:resource=
               "#ReserveHotel_Request"/>
    <hasOutput rdf:resource=
               "#ReserveHotel_Response"/>
    <creates rdf:resource=
             "#HotelReservation"/>
  </operation>

  <Message rdf:about=
           "#ReserveHotel_Request">
    <hasPart rdf:resource="#HotelChain"/>
    <hasPart rdf:resource="#StartDate"/>
    <hasPart rdf:resource="#NumNights"/>
    <hasPart rdf:resource="#NumPersons"/>
    <hasPart rdf:resource="#NumRooms"/>
    <hasPart rdf:resource="#FirstName"/>
    <hasPart rdf:resource="#LastName"/>
  </Message>

  <Message rdf:about=
           "#ReserveHotel_Response">
    <hasPart rdf:resource=
             "#HotelReservation"/>
  </Message>

  <Condition rdf:about="#greaterThanToday">
    <hasConcept rdf:resource="#greaterThan"/>
    <onPart rdf:resource="#StartDate"/>
    <hasValue rdf:resource="#TodaysDate"/>
  </Condition>

  <!-- Similarly we can define Condition
       #greaterThanZero on parts #NumRooms
       and #NumNights -->

  <QualifiedConcept rdf:about="#HotelChain">
    <isA rdf:resource="#Chain"/>
    <ofKind rdf:resource="#Hotel"/>
  </QualifiedConcept>

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

  <QualifiedConcept rdf:about="#TodaysDate">
    <isA rdf:resource="#Date"/>
    <ofKind rdf:resource="#Today"/>
  </QualifiedConcept>

  <!-- Similarly we can define Qualified
       Concepts for #NumNights, #NumPersons,
       #NumRooms, #FirstName and #LastName -->

  <BasicConcept rdf:about="#Hotel">
    <isA rdf:resource="&wn;hotel"/>
  </BasicConcept>

  <BasicConcept rdf:about="#Chain">
    <isA rdf:resource="&wn;chain"/>
  </BasicConcept>

  <BasicConcept rdf:about="#Start">
    <isA rdf:resource="&wn;start"/>
  </BasicConcept>

  <BasicConcept rdf:about="#Date">
    <isA rdf:resource="&wn;date"/>
  </BasicConcept>

  <BasicConcept rdf:about="#greaterThan">
    <isA rdf:resource="&wn;greater_than"/>
  </BasicConcept>

  <BasicConcept rdf:about="#Date">
    <isA rdf:resource="&wn;date"/>
  </BasicConcept>

  <BasicConcept rdf:about="#Today">
    <isA rdf:resource="&wn;today"/>
  </BasicConcept>

  <BasicConcept rdf:about="#Reservation">
    <isA rdf:resource="&wn;reservation"/>
  </BasicConcept>

  <!-- Similarly we can define Basic
       Concepts for #nights, #rooms,
       #number, #persons, #name, etc. -->

</definitions>
```

A Book-Buying Service example is presented in [15].

6 Theory of Substitution of Services

Next, we will investigate the theoretical aspects of USDL. This involves concepts from set theory. From a systems integration perspective, an engineer is interested in finding (discovering) a service that accomplishes some necessary task. Of course, such a service may not be present in any service directory. In such a case the discovery software should return a set of services that can be used in a context expecting a service that meets that description (of course, this set may be empty). To find services that can be substituted for a given service that is not present in the directory, we need to develop a theory of *service substitutability*. We develop such a theory in this section. Our theory relates service substitutability to WordNet's semantic relations.

In order to develop this theory, we must first formally define constructs such as USDL-described concepts, affects, conditions and services, which we will also call concepts, affects, conditions and services for short. While it is possible to work directly with the XML USDL syntax, doing so is cumbersome and so we will instead opt for set theoretic notation.

Definition 1 (Set of WordNet Lexemes)

Let Ω be the set of WordNet lexemes. The following semantic relations exist on elements of Ω.

1. **Synonym**: A pair of WordNet Lexemes having the same or nearly the same meaning have the synonym relation. Example, 'purchase' is a synonym of 'buy'.
2. **Antonym**: A pair of WordNet Lexemes having the opposite meaning have the antonym relation. Example, 'start' is an antonym of 'end'.
3. **Hyponym**: A word that is more specific than a given word is called the subordinate or hyponym of the other. Example, 'car' is a hyponym of 'vehicle'.
4. **Hypernym**: A word that is more generic than a given word is called the super-ordinate or hypernym of the other. Example, 'vehicle' is a hypernym of 'car'.
5. **Meronym**: A word that names a part of a larger whole is a meronym of the whole. Example, 'roof' and 'door' are meronyms of 'house'.
6. **Holonym**: A word that names the whole of which a given word is a part is a holonym of the part. Example, 'house' is a holonym for 'roof' and 'door'.

Definition 2 (Representation of USDL Concepts)

1. A Basic Concept $c = x$, where x is a WordNet lexeme, defines the values of *isA* property. Example, *customer* is a Basic Concept and a WordNet lexeme.
2. A Qualified Concept $c = (X, Y)$, where X, Y are USDL concepts, defines the values of *isA* and *ofKind* properties. Example, concept *FlightNumber* is a *number* of kind *flight* represented as (*number*, *flight*).
3. An Inverted Concept c is represented as $\neg X$ where X is an USDL concept. Example, concept *not a customer name* can be represented as $\neg$(*name*, *customer*).
4. Let X, Y be USDL Concepts.
   (i) Conjunctive Concept c is represented as $X \wedge Y$. Example, concept *EvenRationalNumber* is represented as (*number*, *even*) $\wedge$ (*number*, *rational*).
   (ii) Disjunctive Concept c is represented as $X \vee Y$. Example, concept *OrderNumber/AvailabilityMessage* is represented as (*number*, *order*) $\vee$ (*message*, *availability*).

Definition 3 (Universe of USDL Concepts)

Let Θ be the set of USDL concepts. Θ can be inductively constructed as follows:

1. $x \in \Omega$ implies $x \in \Theta$
2. $X, Y \in \Theta$ implies $(X, Y) \in \Theta$
3. $X \in \Theta$ implies $\neg X \in \Theta$
4. $X, Y \in \Theta$ implies $X \vee Y \in \Theta$
5. $X, Y \in \Theta$ implies $X \wedge Y \in \Theta$

Definition 4 (Semantic relations of Basic Concepts)

Semantic relations hold between two Basic concepts if their corresponding WordNet lexemes have the same semantic relation in Ω. For example, Basic Concept *Purchase* is a synonym of Basic Concept *Buy*.

Definition 5 (Synonym and Antonym relation of Qualified Concepts)

Let C_1 and C_2 are Qualified Concepts where C_1=(X_1, Y_1), C_2=(X_2, Y_2) and X_1, X_2, Y_1, $Y_2 \in \Theta$.

1. C_1 is synonym of C_2 if X_1 is recursively a synonym of X_2 and Y_1 is recursively a synonym of Y_2.
2. If $X_1 = w_1$ and $X_2 = w_2$ where w_1, $w_2 \in \Omega$, then X_1 is synonym of X_2 if w_1 and w_2 have the synonym relation in Ω.

For example, Qualified Concept (*date, begin*) is a synonym of Qualified Concept (*date, start*). Similarly we can determine the antonym relation between Qualified Concepts. For example, Qualified Concept (*date, begin*) is a antonym of Qualified Concept (*date, end*).

Definition 6 (Hyponym and Hypernym relation of Qualified Concepts)

Let C_1 and C_2 are Qualified Concepts where C_1=(X_1, Y_1), C_2=(X_2, Y_2) and X_1, X_2, Y_1, $Y_2 \in \Theta$.

1. C_1 is hypernym of C_2 if any one of the following holds:
   (i) X_1 is recursively a hypernym of X_2 and Y_1 is recursively a hypernym of Y_2.
   (ii) X_1 is recursively a hypernym of X_2 and Y_1 is recursively a synonym of Y_2.
   (iii) X_1 is recursively a synonym of X_2 and Y_1 is recursively a hypernym of Y_2.
2. If $X_1 = w_1$ and $X_2 = w_2$ where w_1, $w_2 \in \Omega$, then X_1 is hypernym of X_2 if w_1 and w_2 have the hypernym relation in Ω.

For example, Qualified Concept (*number, vehicle*) is a hypernym of Qualified Concept (*number, car*). Similarly we can determine the hyponym relation of Qualified Concepts. For example, Concept (*number, car*) is a hyponym of (*number, vehicle*).

Definition 7 (Holonym and Meronym relation of Qualified Concepts)

Let C_1 and C_2 are Qualified Concepts where C_1=(X_1, Y_1), C_2=(X_2, Y_2) and X_1, X_2, Y_1, $Y_2 \in \Theta$.

1. C_1 is meronym of C_2 if any one of the following holds:
   (i) X_1 is recursively a meronym of X_2 and Y_1 is recursively a meronym of Y_2.
   (ii) X_1 is recursively a meronym of X_2 and Y_1 is recursively a synonym of Y_2.
   (iii) X_1 is recursively a synonym of X_2 and Y_1 is recursively a meronym of Y_2.
2. If $X_1 = w_1$ and $X_2 = w_2$ where w_1, $w_2 \in \Omega$, then X_1 is meronym of X_2 if w_1 and w_2 have the meronym relation in Ω.

For example, Qualified Concept (*door, brown*) is a meronym of Qualified Concept (*house, brown*). Similarly we can determine the holonym relation of Qualified Concepts. For example, Qualified Concept (*house, brown*) is a holonym of Qualified Concept (*door, brown*).

Definition 8 (Semantic relations between Inverted Concepts)

Let C_1 and C_2 be two Inverted concepts where $C_1 = \neg X_1$ and $C_2 = \neg X_2$.

1. C_1 is a synonym of C_2 if X_1 and X_2 are synonyms.
2. C_1 is an antonym of C_2 if X_1 and X_2 are antonyms.
3. C_1 is a hypernym of C_2 if X_1 and X_2 are hyponyms and vice versa.
4. C_1 is a meronym of C_2 if X_1 and X_2 are holonyms and vice versa.

For example, Inverted Concept $\neg$(*date, begin*) is a synonym of Inverted Concept $\neg$(*date, start*). The synonym-antonym relation, hyponym-hypernym relation and meronym-holonym relation can be extended to Conjunctive and Disjunctive concepts.

Definition 9 (Semantic relations between Conjunctive (resp., Disjunctive) Concepts)

Let C_1 and C_2 be two Conjunctive (resp., Disjunctive) concepts where $C_1 = X_1 \wedge Y_1$ and $C_2 = X_2 \wedge Y_2$.

1. C_1 is a synonym of C_2 if X_1 is a synonym of X_2 and Y_1 is a synonym of Y_2 OR X_1 is a synonym of Y_2 and Y_1 is a synonym of X_2.
2. C_1 is a hypernym of C_2 if one of the following holds:
   (i) X_1 is a hypernym of X_2 and Y_1 is a hypernym/synonym of Y_2
   (ii) X_1 is a hypernym/synonym of X_2 and Y_1 is a hypernym of Y_2
   (iii) X_1 is a hypernym of Y_2 and Y_1 is a hypernym/synonym of X_2.
   (iv) X_1 is a hypernym/synonym of Y_2 and Y_1 is a hypernym of X_2.

For example, Conjunctive Concept (*vehicle, blue*) $\wedge$ (*vehicle, automatic*) is a hypernym of (*car, blue*) $\wedge$ (*car, automatic*). Similar to the above defined hypernym relation, we can define the antonym, hyponym, meronym, and holonym relations between Conjunctive (resp., Disjunctive) Concepts.

Definition 10 (Substitution of Concepts)

1. **Exact Substitution**: For any concepts $C, C' \in \Theta$, if C is a synonym of C', then C is the exact substitutable of C' and C can safely be used in a context expecting concept C'. Example, concept *Purchase* is an exact substitutable of concept *Buy*.
2. **Generic Substitution**: For any concepts $C, C' \in \Theta$, if C is a hypernym of C', then C is the generic substitutable of C' and C can safely be used in a context expecting concept C' or a super-ordinate of C'. Example, concept (*number, vehicle*) is a generic substitutable of concept (*number, car*).
3. **Specific Substitution**: For any concepts $C, C' \in \Theta$, if C is a hyponym of C', then C is the specific substitutable of C' and C can safely be used in a context expecting concept C' or a sub-ordinate of C'. Example, concept (*number, car*) is a specific substitutable of concept (*number, vehicle*).
4. **Part Substitution**: For any concepts $C, C' \in \Theta$, if C is a meronym of C', then C is the part substitutable of C' and C can safely be used in a context expecting a concept that is a part of C'. Example, concept *Roof* is a part substitutable of concept *House*.

5. **Whole Substitution**: For any concepts $C, C' \in \Theta$, if C is a holonym of C', then C is the whole substitutable of C' and C can safely be used in a context expecting a concept that is a whole of C'. Example, concept *House* is a whole substitutable of concept *Roof*.

Definition 11 (Representation of Affects)

Let $\Gamma = \{(L, E) \mid L \in (\Psi \cup \Theta), \mathrm{E} \in \Theta\}$ be the set of USDL side-effects, where $\Psi = \{creates, updates, deletes, finds\}$, L is the affect type and E is the affected object. The affect type could be one of the pre-defined affects from Ψ or a generic effect which is described as a concept.

Definition 12 (Substitution of Affects)

USDL affect is represented as a pair where the first element is the affect type and second element is the affected object. Both affect type and the affected object are described as USDL concepts.

Let A_1 and A_2 be two affects where $A_1 = (L_1, E_1)$ and $A_2 = (L_2, E_2)$. A_1 can safely be used in a context expecting affect A_2 if all of the following hold:

1. Concept L_1 is substitutable for L_2
2. Concept E_1 is substitutable for E_2.

These substitutables can be of kind Exact, Generic, Specific, Part, or Whole which also determines the kind of substitution of the affect A_1 in a context expecting A_2. Example, affect (*finds*, *VehicleNumber*) is a generic substitution of affect (*lookup*, *CarNumber*) as concept *finds* is an exact substitutable of *lookup* and concept *VehicleNumber* is a generic substitutable of concept *CarNumber*.

Definition 13 (Representation of Conditions)

Let $\Phi = \{(P, Arg_1, Arg_2) \mid \mathrm{P}, Arg_1, Arg_2 \in \Theta\}$ be the set of USDL conditions. P is the constraint which is either a binary or a unary predicate. Arg_1 is the concept on which the predicate acts and Arg_2 is the concept which represents a value. Arg_2 is an optional parameter.

Definition 14 (Substitution of Conditions)

USDL condition is represented as a tuple made up of the constraint or predicate and two arguments. The constraint and the arguments are described as USDL concepts.

Let C_1 and C_2 be two conditions where $C_1 = (P_1, FirstArg_1, SecondArg_1)$ and $C_2 = (P_2, FirstArg_2, SecondArg_2)$. C_1 can safely be used in a context expecting condition C_2 if all of the following hold:

1. Concept P_1 is substitutable for P_2
2. Concept $FirstArg_1$ is substitutable for $FirstArg_2$
3. Concept $SecondArg_1$ is substitutable for $SecondArg_2$.

These substitutables can be of kind Exact, Generic, Specific, Part, or Whole which also determines the kind of substitution of the condition C_1 in a context expecting C_2. Example, condition (*greaterThan*, *NumberOfNights*, 0) is an exact substitution of condition (*moreThan*, *NumberOfNights*, 0).

Definition 15 (Representation of a Web Service)

For any set S, let $S^* = \{\epsilon \mid \epsilon \notin S\} \cup \{(x, y) \mid x \in S, y \in S^*\}$ be the set of lists over S.
Let Σ be the set of USDL service descriptions represented in the form of terms. The USDL description of a web service consists of

1. A list of Inputs, $I \in \Theta^*$
2. A list of Outputs, $O \in \Theta^*$
3. A list of Pre-Conditions, *Pre-Condition* $\in \Phi^*$
4. A list of Post-Conditions, *Post-Condition* $\in \Phi^*$
5. Side-effects, *(affect-type, affected-object)* $\in \Gamma$

USDL service description can be treated as a term of first-order logic [16]. The side-effect of a service comprises of an affect type and the affected object. The service can be converted 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.

We represent services as triples so that they can be treated as terms in first-order logic. The first-order logic unification algorithm [16] then can be extended to specialized unifications for exact, generic, specific, part and whole substitutions. This work is in progress [10].

Now that the formal definitions of concept, affects, conditions and service descriptions are given, we would like to extend the theory of substitutability over Σ so that we can reason about substitutability of services.

Definition 16 (Substitution of a Web Service)

Let σ and σ' be two services where
σ is represented as *(Pre-Condition, affect-type(affected-object, I, O), Post-Condition)* and
σ' is represented as *(Pre-Condition′, affect-type′(affected-object′, I′, O′), Post-Condition′).*
σ can safely be used in a context expecting service σ' if all of the following hold:

1. *Pre-Condition* is substitutable for *Pre-Condition′*
2. If the terms *affect-type(affected-object, I, O)* and *affect-type′(affected-object′, I′, O′)* can be unified by applying an extended unification algorithm. The unification mechanism applied is different based on the kind of substitution needed.
3. *Post-Condition* is substitutable for *Post-Condition′*

Definition 17

For any services $\mathcal{S}_1$, $\mathcal{S}_2 \in \Sigma$, we say $\mathcal{S}_1 \preceq \mathcal{S}_2$ if $\mathcal{S}_1$ is a substitutable of $\mathcal{S}_2$ based on one of the WordNet semantic relations.

Thus far our notions of service substitutability are based on the six WordNet semantic relations discussed earlier. However, one can define the notion of service substitutability independently using the actual semantics (e.g., denotational semantics) of the program that realizes this service. Consider a service $\mathcal{S}_1$ with inputs $\mathcal{I}_1$ and outputs $\mathcal{O}_1$, and another service $\mathcal{S}_2$ with inputs $\mathcal{I}_2$ and outputs $\mathcal{O}_2$; we ignore the side-effects of these services for the moment. The ideal conditions under which service $\mathcal{S}_1$ can be substituted for service $\mathcal{S}_2$ is the following: $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2$ and $\mathcal{O}_1 \sqsupseteq \mathcal{O}_2$.

Essentially, the inputs needed by $\mathcal{S}_1$ must be present in the inputs being provided in anticipation of availability of $\mathcal{S}_2$. Likewise, the outputs produced by service $\mathcal{S}_1$ should contain the outputs anticipated from service $\mathcal{S}_2$. In such a case, $\mathcal{S}_1$ can be directly substituted for $\mathcal{S}_2$. There can be other types of general substitution relation defined. However, for these other types of substitutions, the code of the service being used for substitution may have to be modified or wrappers placed around it.

One can, however, develop a more general notion of substitutability based on denotational semantics [17]. Let $[\![\mathcal{S}_1]\!]$ and $[\![\mathcal{S}_2]\!]$ be the semantic denotations of programs that implement services $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively (note that the side-effects of these services will be captured as the state that becomes an argument in a denotational definition). Note that $[\![\mathcal{S}_1]\!]$ and $[\![\mathcal{S}_2]\!]$ can be regarded as points in a complete partial order [17] that represents the space of all functions. Service $\mathcal{S}_1$ can be substituted for $\mathcal{S}_2$ if $[\![\mathcal{S}_1]\!]$ and $[\![\mathcal{S}_2]\!]$ lie in the same chain in the complete partial order (i.e., either $[\![\mathcal{S}_1]\!] \sqsubseteq [\![\mathcal{S}_2]\!]$ or $[\![\mathcal{S}_2]\!] \sqsubseteq [\![\mathcal{S}_1]\!]$ where $\sqsubseteq$ is the relation that induces the complete partial order among denotations of the services).

Given the definition of substitutability based on denotational semantics, one can prove the soundness and completeness of our notion of substitutability based on the WordNet semantic relations, i.e.,

$\mathcal{S}_1 \preceq \mathcal{S}_2 \Rightarrow [\![\mathcal{S}_1]\!] \sqsubseteq [\![\mathcal{S}_2]\!]$ (soundness)

$[\![\mathcal{S}_1]\!] \sqsubseteq [\![\mathcal{S}_2]\!] \Rightarrow \mathcal{S}_1 \preceq \mathcal{S}_2$ (completeness)

Intuitively, one can see that these relationships hold, since the $\preceq$ relation is defined in terms of subsumption of terms describing the service's inputs and outputs and its effect (create, update, delete, find and the generic affects). These soundness and completeness proofs are not included here due to lack of space.

7 Service Discovery

Now that our theory of service substitutability has been developed, it can be used to build tools for automatically discovering services as well as for automatically composing them. It can also be used to build a service search engine that discovers matching services and ranks them.

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 to look for a "matching" service.

A discovery engine gets USDL descriptions from a service directory and converts them into terms of logic. The terms corresponding to the USDL query can be compared with the terms from the directory using an extended/special unification algorithm. Depending on the type of match required, the unification mechanism could be different. That is, the matching or unification algorithm used can look for an *exact*, *generic*, *specific*, *part* or a *whole* match depending on the desire of the user. Part and Whole substitutions are not useful while looking for matching services, but are very useful while selecting services for service composition. Also using Part or Whole substitutions for discovery may produce undesired side-effects.

The discovery engine can also rank the various services discovered. In this scenario, the discovery engine returns a list of substitutable services after applying ranking based on the kind of match obtained. Exact substitutables are assigned the highest rank among the different kind of substitutables. The following is the default ranking order used for the different substitutions.

1. Exact Substitution: The matching service obtained is equivalent to the service in the query.
2. Generic Substitution: The matching service obtained subsumes the service in the query.
3. Specific Substitution: The matching service obtained is subsumed by the service in the query.
4. Whole Substitution: The matching service obtained is a composite service of the service in the query and some other services.
5. Part Substitution: The matching service obtained is a part of a composite service that the query describes.

The development of a service discovery engine based on these ideas is in progress.

With the USDL descriptions and query language in place, numerous applications become possible ranging from querying a database of services to rapid application development via automated integration tools and even real-time service composition [12]. Take our flight reservation service example. Assume that somebody wants to find a travel reservation service and that they query a USDL database containing general purpose flight reservation services, bus reservation services, etc. One could then form a USDL query consisting of a description of a travel reservation service and the database could respond with a set of travel reservation services whether it be via flight, bus, or some other means of travel. This flexibility of generalization and specialization is gained from semantic information provided by USDL.

8 Service Composition

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 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 postconditions of $\mathcal{S}_1$ must imply the preconditions of $\mathcal{S}_2$.

In fact, the WordNet Universal ontology can also be helpful in automatically discovering services that can be composed together to satisfy a service discovery query. To achieve this, the discovery engine looks at the USDL concepts that describe the service in the query. It then searches the WordNet ontology to find out the meronymous components of that concept. The services that exactly match the meronymous components are then discovered using the standard discovery mechanism. Preconditions and postcondition consistency is then used to find the order in which the meronymous components should be stitched together to produce the desired service.

A service composition engine of this kind is under development. Such an engine can also aid a systems integrator in rapidly creating composite services, i.e., services consisting of the composition of already existing services. In fact, such an engine can also be extended to automatically generate boilerplate code to manage the composite service, as well as menial inter-service data format conversions needed to glue the meronymous components together.

9 Comparison with OWL-S, WSDL-S, and WSML

In this section we present a comparison of USDL with other popular approaches such as OWL-S [5], WSML [1], and WSDL-S [24]. Our goal is 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 [9]. 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 aforementioned semantic aliasing problem [9]. Other recent approaches such as WSMO, WSML, WSDL-S, etc., suffer from the same limitation [1]. 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 [10]). 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 availabil-

ity 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 *Reserve-Flight* 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.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:process="http://www.daml.org/services/owl-s/1.0/Process.owl#">
<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:AtomicProcess rdf:about="#BookFlight"/>
    </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.

10 Related Work

Discovery and composition of web services has been active area of research recently [18, 19, 20, 21, 22, 23]. 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, hyponym, hypernym, 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 [13]. 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 [12].

11 Conclusions and Future Work

To reliably catalogue, search and compose services in a semi-automatic to fully-automatic manner we need standards to publish and document services. This requires language standards for specifying not just the syntax, i.e., prototypes of service procedures and messages, but it also necessitates a standard formal, yet high-level means for specifying the semantics of service procedures and messages. We have addressed these issues by defining a universal service-semantics description language, its semantics, and we have proved some useful properties about this language. The current version of USDL incorporates current standards in a way to further aid markup of IT services by allowing constructs to be given meaning in terms of an OWL based WordNet ontology. This approach is more practical and tractable than other approaches because description documents are more easily created by humans and more easily processed by computers. USDL is currently being used to formally describe web-services related to emergency response functions [11].

Our current and future work involves the application of USDL to formally describing commercial service repositories (for example SAP Interface Repository and services listed in UDDI), as well as to service discovery and rapid application development (RAD) in commercial environments [12]. Current and future work also includes automatically generating USDL description from the code/documentation of a service [12] as well developing tools that will allow automatic generation of new services based on combining USDL descriptions of existing atomic services. The interesting problem that arises then: can USDL description of such automatically generated services be also automatically generated? This problem is also part of our current/future work.

References

[1] A conceptual comparison between WSMO and OWL-S. `www.wsmo.org/TR/d4/d4.1/v0.1`.

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

[3] Resource Description Framework. `http://www.w3.org/RDF`.

[4] SAP Interface Repository. `http://ifr.sap.com/catalog/query.asp`.

[5] Semantic markup for web services. `www.daml.org/services/owl-s/1.0/owl-s.html`.

[6] Web Ontology Language Reference. `http://www.w3.org/TR/owl-ref`.

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

[8] WordNet: A Lexical Database for the English Language. `www.cogsci.princeton.edu/~wn`.

[9] S. Balzer, T. Liebig, and M. Wagner. Pitfalls of OWL-S - a practical semantic web use case. In *ICSOC*, 2004.

[10] S. Kona, A. Bansal, G. Gupta, and T. Hite. Automatic Service Discovery and Composition with USDL. Working paper, 2006.

[11] 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.

[12] T. Hite. Service Composition and Ranking: A strategic overview. Internal Report, Metallect Inc., 2005.

[13] R. Hull and J. Su. Tools for design of composite web services. In *SIGMOD*, 2004.

[14] L. Simon, A. Bansal, A. Mallya, S. Kona, G. Gupta, and T. Hite. Towards a Universal Service Description Language. In *Next Generation Web Services Practices*, pp. 175-180, 2005.

[15] 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.

[16] J.W. Lloyd. Foundations of Logic Programming. Springer-Verlag, 1987.

[17] D. Schmidt. Denotational Semantics: A Methodology for Language Development. 1986.

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

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

[20] S. McIlraith, T.C. Son Adapting golog for composition of semantic web services. In *KRR*, pages 482–493, 2002.

[21] S. McIlraith, S. Narayanan Simulation, verification and automated composition of web services. In *World Wide Web Conference*, 2002.

[22] G. Picinielli, et al. Web service interfaces for inter-orgranizational business processes - an infrastructure for automated reconciliation. In *EDOC*, pages 285-292, 2002.

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

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