A Universal Service-Semantics Description Language

Luke Simon, Ajay Bansal, Ajay Mallya, 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

To fully utilize web-services, users and applications should be able to discover, deploy, compose and synthesize services automatically. This automation can take place only if a formal semantic description of the web-services is available. In this paper we present the design of 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 program documentation that will allow sophisticated conceptual modeling and searching of available web-services, automated service composition, and other forms of automated service integration. The design of USDL is presented, along with examples, and its formal semantics given. A theory of service composition for USDL is presented and proved sound and complete.

1 Introduction

The next milestone in the Web's evolution is making *services* ubiquitously available. 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. 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. For web-services to become practical, an infrastructure needs to be supported that allows users to discover, deploy, compose, and synthesize services automatically. Such an infrastructure *must* be semantics based so 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 this infrastructure. These efforts include approaches based on the semantic web (such as OWL-S [4]) as well as those based on XML, such as Web Services Description Language (WSDL). Approaches such 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. 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 program 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 to check if this exact service is available, and if not available, then whether it can be synthesized by composing two or more services listed in this (or another) directory.

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 WordNet OWL a suitable candidate for a universal ontology of atomic concepts upon which arbitrary meets and joins can be added in order to gain tractable flexibility.

In the next section we describe what is meant by conceptual modeling and how such a thing could be obtained via a common universal ontology based on WordNet. 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, and in addition, a short example WSDL service is annotated, so that a concrete example of instances of the various OWL classes and properties can be seen. Section 5 gives a complete USDL annotation for a BookBuying service. Automatic discovery and composition of web services using USDL is discussed in section 6. In section 7 we explore some of the theoretical aspects of service description in USDL. Comparison of USDL with other approaches like OWL-S and WSML is discussed in section 8. Finally, other related work and conclusion are given in the remaining sections.

2 A Universal Ontology

We can describe what any given computer program does in a logical manner, from first-principles. 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 software is needed in order to verify it. However, such solutions are both too low-level and too tedious (and not to mention, 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 point of the end user or integrator of the service, as opposed to the far more difficult questions, such as, what does the program do from a computational view? The distinction is subtle, but is a distinction of granularity as well as a distinction of scope: 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 program to model these concepts.

At some point, a common denominator must be agreed upon in order to allow interoperability and machine-readability of our documents. 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 tertiary 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. This problem can be overcome by standardizing on a sufficiently comprehensive set of atomic concepts, i.e. a universal ontology, along with restricted connectives, such that the semantics always equate things that are conceptually equal.

Another approach would be to use industry specific ontologies along with OWL (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 then how will new innovative services in new upcoming industries be ascribed formal meaning? Of course, a universal ontology will have no difficulty in describing such new services.

Thus, our common conceptual ground must be

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 [7]. 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 [1]. 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 an 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. Finally, USDL restricts its conceptual constructors, in order to address the problems of semantic aliasing and tractability.

3 USDL: An Overview

Like WSDL, USDL describes a service in terms of ports and messages. The semantics of the service is given using the WordNet OWL ontology. USDL maps ports (operations provided by the service) and messages (operation parameters) to disjunctions of conjunctions of (possibly negated) concepts in the WordNet OWL 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 the following operations: *create, update, delete,* or *find*, but also allows for a generic *affects* side-effect when none of the others apply. An application that wishes to make use of a service automatically should be able to reason with WordNet atoms using the WordNet OWL ontology.

USDL is perhaps the first language that attempts 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 [4]. As mentioned earlier, WSDL only defines syntax of the service; USDL can be thought of as providing the missing semantic component. USDL can be thought of as a formal language for program documentation. Thus instead of documenting the function of a service as comments in English, one writes 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 complimentary. OWL-S primarily describes the states that exists before and after the service and how a service is composed of other smaller sub-services (if any). Description of atomic services is left underspecified 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 complimentary 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 that are described using USDL.

In order to develop a theory for composing services, we also define the formal semantics of USDL. The syntactic terms describing ports and messages are mapped to disjunctions and conjunctions of (possibly negated) OWL WordNet ontological terms. These disjunctions and conjunctions are represented by points in the lattice obtained from the WordNet ontology with regards to the OWL subsumption relation. A service is then formally defined as a function, labeled with zero or more side-effects, between points in this lattice.

The main contribution of our work is the design of a universal service- semantics description language USDL, along with its formal semantics, and soundness and completeness proofs for a theory of service composition with USDL.

4 Design of USDL

The design of USDL rests on two formal languages: Web Services Description Language (WSDL) [6] and Web Ontology Language (OWL) [5]. The Web Services Description Language (WSDL) [6], 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) [5], was developed as an extension to the Resource Description Framework (RDF) [2], 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.

As mentioned earlier, 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 ports. These constructs are actually aggregates of service components which will actually be directly ascribed meaning. Messages consist of typed parts and ports consist 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. Semantically, instances of *Concept* form a complete lattice, which will be covered in section 7.

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

  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty
       rdf:resource="#hasCondition"/>
      <owl:mincardinality
       rdf:datatype="&xsd;nonNegativeInteger">
         0
      </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 USDL *Concept* class denotes the top element of the lattice of 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 atomic concepts. These subclasses of *Concept* are

- AtomicConcept
- InvertedConcept
- ConjunctiveConcept
- DisjunctiveConcept

4.1.1 Atomic Concept

AtomicConcept is the actual contact point between USDL and WordNet. This class acts as proxy for WordNet lexical concepts.

```
<owl:Class rdf:about="#AtomicConcept">
  <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:mincardinality
       rdf:datatype="&xsd;nonNegativeInteger">
         0
      </owl:mincardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  ...
</owl:Class>
```

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

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

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

4.1.2 Inverted Concept

In the case of *InvertedConcept* the corresponding semantics are the complement of the WordNet lexical 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>
```

4.1.3 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 *AtomicConcepts*, *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 as a conjunction of these types of effects. For those services that cannot be described in terms of a combination of these specializations, the parent *affects* property can be used instead, or the property can be omitted entirely when the meaning of the operation parameter messages are enough for conceptual reasoning. The purpose of limiting the types of services as opposed to allowing the creation of new arbitrary side-effect types, for example via OWL-DL, is to: (i) make USDL more structured and therefore easier to create documents in, (ii) make USDL computationally more tractable for programs that process large volumes of USDL documents, and (iii) help prevent the semantic aliasing problem mentioned in section 2.

4.3 Conditions and Constraints

Services may have some external 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 atmost 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 [3], which makes use of the following message.

```
<message name="&flight;ReserveFlight_Request">
  <part name="&flight;CustomerName"
        type="xsd:string">
  <part name="&flight;FlightNumber"
        type="xsd:string">
  <part name="&flight;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:about="#Message">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasPart"/>
      <owl:minCardinality
       rdf:datatype="&xsd;nonNegativeInteger">
         0
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</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 *Itinerary* 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=
         "&flight;ReserveFlight_Request">
  <hasPart>
    <AtomicConcept
     rdf:about="&flight;CustomerName">
      <isA rdf:resource="&wn;name"/>
      <ofKind>
        <AtomicConcept>
          <isA rdf:resource="&wn;customer"/>
        </AtomicConcept>
      </ofKind>
    </AtomicConcept>
  </hasPart>

  <hasPart>
    <AtomicConcept
     rdf:about="&flight;FlightNumber">
      <isA rdf:resource="&wn;number"/>
      <ofKind>
        <AtomicConcept>
          <isA rdf:resource="&wn;flight"/>
        </AtomicConcept>
      </ofKind>
    </AtomicConcept>
  </hasPart>
  ...
</Message>
```

4.5 Ports

Services consist of ports, which are collections of procedures or operations that are parametric on messages. Our example flight reservation service might contain a port definition for a flight reservation service that takes as input an itinerary and outputs a reservation receipt.

```
<portType name="&flight;Flight_Reservation">
  <operation name="&flight;ReserveFlight">
    <input message=
           "&flight;ReserveFlight_Request"/>
    <output message=
            "&flight;ReserveFlight_Response"/>
  </operation>
  ...
</portType>
```

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

```
<owl:Class rdf:about="#Port">
  <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="#Port"/>
  <rdfs:range rdf:resource="#Operation"/>
</owl:ObjectProperty>
```

As with the case of messages, ports are not directly assigned meaning via the OWL WordNet ontology. Instead the individual *Operation*s of a port 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="#affects"/>
      <owl:minCardinality
       rdf:datatype="&xsd;nonNegativeInteger">
         0
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

An operation can have multiple or no values for the *affects* property, all of which are of type USDL *Concept*, which is the target of the effect.

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

Finishing our flight reservation service example, we can describe the main side-effect of invoking the `reserve` operation in USDL as follows.

```
<Port rdf:about="&flight;Flight_Reservation">
  <hasOperation>
    <Operation rdf:about="&flight;ReserveFlight">
      <creates>
        <AtomicConcept
         rdf:about="flight_reservation">
          <isA rdf:resource="&wn;reservation"/>
          <ofKind>
            <AtomicConcept>
              <isA rdf:resource="&wn;flight"/>
            </AtomicConcept>
          </ofKind>
        </AtomicConcept>
      </creates>
    </Operation>
  </hasOperation>
  ...
</Port>
```

Again, note that it is not necessary to annotate the operation with regards to its input and output parameters, as these are already annotated by *Message* surrogates.

5 Semantic Description of a Service

A simple Book Buying Service: The service described here is a simplified book buying 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 preconditions, the service provides certain outputs and has specific effects.

The service has an input pre-condition that a User Identifier for the buyer must exist before invoking the service. It also has a global constraint that a valid credit card number for the buyer must exist.

5.1 WSDL definition

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

```
<definitions>
  ...
  <portType name="BookBuying_Service">
    <operation name="BookBuying">
      <input message="BuyBook_Request"/>
      <output message="BuyBook_Response"/>
    </operation>
  </portType>

  <message name="BuyBook_Request">
    <part name="BookISBN"
          type="xsd:string"/>
    <part name="UserIdentifier"
          type="xsd:string"/>
    <part name="Password"
          type="xsd:string"/>
  </message>

  <message name="BuyBook_Response">
    <part name="OrderNumber/Availability"
          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>
  ...
  <port rdf:about="#BookBuying_Service">
    <hasOperation>
      <operation name="BuyBook">
        <creates>
          <AtomicConcept rdf:about="#BookOrder">
            <isA rdf:resource="&wn;order"/>
            <ofKind>
              <AtomicConcept>
                <isA rdf:resource="&wn;book"/>
              </AtomicConcept>
            </ofKind>
            <hasCondition>
```

```
            <Condition rdf:about="#exists">
              <AtomicConcept>
                <isA rdf:resource="&wn;exists"/>
              </AtomicConcept>
              <onPart rdf:about="#CreditCard">
                <AtomicConcept>
                  <isA rdf:resource="&wn;card"/>
                  <ofKind>
                    <AtomicConcept>
                   <isA rdf:resource=
                            "&wn;credit"/>
                 </AtomicConcept>
                    </ofKind>
                </AtomicConcept>
              </onPart>
            </Condition>
          </hasCondition>
        </AtomicConcept>
      </creates>
    </operation>
  </hasOperation>
</port>

<portType name="BookBuying_Service">
  <operation name="BuyBook">
    <input  message="BuyBook_Request"/>
    <output message="BuyBook_Response"/>
  </operation>
</portType>

<Message rdf:about="#BuyBook_Request">
  <hasPart>
    <AtomicConcept rdf:about="#BookISBN">
      <isA rdf:resource="&wn;identifier"/>
      <ofKind>
        <AtomicConcept>
          <isA rdf:resource="&wn;book"/>
        </AtomicConcept>
      </ofKind>
    </AtomicConcept>
  </hasPart>

  <hasPart>
    <AtomicConcept rdf:about=
                  "#UserIdentifier">
      <isA rdf:resource="&wn;identifier"/>
      <ofKind>
        <AtomicConcept>
          <isA rdf:resource="&wn;user"/>
        </AtomicConcept>
      </ofKind>
      <hasCondition>
        <Condition rdf:about="#exists">
          <AtomicConcept>
            <isA rdf:resource="&wn;exists"/>
          </AtomicConcept>
          <onPart rdf:resource=
                  "#UserIdentifier"/>
        </Condition>
      </hasCondition>
    </AtomicConcept>
  </hasPart>

  <hasPart>
      <AtomicConcept>
        <isA rdf:resource="&wn;Password"/>
      </AtomicConcept>
    </hasPart>
  </Message>

  <Message rdf:about="#BuyBook_Response">
    <hasPart>
      <DisjunctiveConcept
       rdf:about="#OrderNumber/Availability">
        <hasConcept>
          <AtomicConcept rdf:about=
                        "#OrderNumber">
            <isA rdf:resource="&wn;number"/>
            <ofKind>
              <AtomicConcept>
                <isA rdf:resource="&wn;order"/>
              </AtomicConcept>
            </ofKind>
          </AtomicConcept>
        </hasConcept>

        <hasConcept>
          <InvertedConcept rdf:about=
                          "#NotAvailable">
            <AtomicConcept>
              <isA rdf:resource="&wn;available"/>
            </AtomicConcept>
          </InvertedConcept>
        </hasConcept>
      </DisjunctiveConcept>
    </hasPart>
  </Message>
  ...
</definitions>
```

6 Service Discovery and Composition

Note that given a directory of services, a USDL description could be included for each service, making the 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. Note that USDL itself could be 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. For matching we could treat USDL descriptions as well as the USDL query as terms, and perhaps use some kind of extended unification to check for a match. This work is currently in progress [9].

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 [14]. Take our flight reservation service example. Assume that some-

body 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 USDL's subtyping relation covered in section 7.

Furthermore, with a pool of USDL services at one's disposal, rapid application development (RAD) tools could be used to aid a systems integrator with the task of creating composite services, i.e. services consisting of the composition of already existing services. The service designer could use such a RAD tool by describing the desired service via a USDL document, and then the tool would query the pool of services for composable sets of services that can be used to accomplish the task as well as automatically generate boilerplate code for managing the composite service, as well as menial inter-service data format conversions and other glue. Of course these additional RAD steps would require other technologies already being researched and developed [10, 11, 16, 13, 15].

At present, we presume only four specific operations (find, create, delete, update); this set of basic operations may be extended as more experience is gained with USDL. In practice, most services, however, deal with manipulating databases and for such services these four operations are sufficient. As stated, one of the reasons for limiting the side-effect types is to safeguard against the semantic aliasing problem described in section 2. This is also one of the main reasons for restricting the combing forms in USDL to conjunction, disjunction, and negated atoms. As discussed in the next section, this allows USDL descriptions to be put into a type of disjunctive normal form, from which a sound and complete notion of subtyping is created.

7 Theory of Service Description

In this section, we will investigate the theoretical aspects of service description via USDL. This involves concepts from set theory, lattice theory, and type theory. From a systems integration perspective, an engineer is interested in finding a set of composable services that accomplish some necessary task. Therefore service description should allow the engineer to describe a service in USDL and receive in return a set of services that can be used in a context expecting a service that meets that description. We prove that this is possible in the soundness and completeness theorems at the end of this section.

In order to prove these theorems, we must first formally define constructs such as USDL described objects and services, which we will also call objects 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. *Let Ω be the set of WordNet nouns and $\leq_\Omega$ be the OWL subsumption relation on Ω.*

Definition 2. *Let Θ be the least set of objects such that:*

1. $x \in \Omega$ implies $x, \neg x \in \Omega$
2. $x, y \in \Theta$ implies $x \cup y, x \cap y \in \Theta$

Hence Θ is simply the set of objects described by USDL concepts.

Definition 3. *Let $\Gamma = \{(L, E) \mid L \in \Psi, E \in \Theta\}$ be the set of USDL side-effects, where $\Psi = \{creates, updates, deletes, finds\}$ L is the affect type and E is the effected object.*

Definition 4. *For any set S, let*

$$S^* = \{\epsilon \mid \epsilon \notin S\} \cup S \cup \{(x, y) \mid x \in S, y \in S^*\}$$

be the set of lists over S. Let $\Sigma = \{(A, I, O) \mid A \in 2^\Gamma, I \in \Theta^*, O \in \Theta^*\}$ be the set of USDL service descriptions, where A is the set of side-effects, I is the list of input parameters, and O^* is the list of output parameters of a particular service.

Now that the formal definitions of object and service descriptions are out of the way, we would like to define a subsumption relation $\leq_\Sigma$ over Σ so that we can reason about composability of services, but this will, in turn, require a subsumption relation $\leq_\Theta$ over Θ. The proof of correctness of $\leq_\Sigma$ is covered by the principle of safe substitution below.

Definition 5. *Assuming without loss of generality that all objects are expressed in disjunctive normal form, then let $\leq_\Theta$ be the ordering relation defined on objects such that:*

1. $x \leq_\Theta y$ for $x, y \in \Omega$ and $x \leq_\Omega y$
2. $\neg x \leq_\Theta \neg y$ if and only if $y \leq_\Omega x$
3. $\bigcup_{\forall i} w_i \leq_\Theta \bigcup_{\forall j} x_j$ if and only if for all $w_j = \bigcap_{\forall k} y_k$ there exists some $x_j = \bigcap_{\forall l} z_l$ such that for every z_l there exists some $y_k \leq_\Omega z_l$.

Definition 6. *Let $(A, I, O) \leq_\Sigma (A', I', O')$ if and only if $\forall (L, E) \in A . \exists (L, E') \in A' . E \leq_{\Theta^*}$ and $I' \leq_{\Theta^*} I$ and $O \leq_{\Theta^*} O'$, where $\leq_{\Theta^*}$ is the element-wise extension of $\leq_\Theta$ to lists of objects.*

Note that $(\Theta, \leq_\Theta)$ and $(\Sigma, \leq_\Sigma)$ respectively form a complete lattice. Given a description of a service $\sigma \in \Sigma$, we can now define the set of composables $C(\sigma)$, which in practice corresponds to a query against a database of services Σ, for services that satisfy description σ. Notice that the definition is contravariant for inputs and covariant for outputs [17]. This contravariance is also seen in the field of type theory with regards to polymorphic subtyping [17], and will be covered in the proof of the principle of safe substitution below.

Definition 7. *Let C be the set of composables parametric over services be a function mapping Σ to a subset of services such that $C(\sigma) = \{\sigma' \mid \sigma' \leq_\Sigma \sigma\}$.*

In order to be able to prove the soundness and completeness of C, we will first need to prove the following lemma known as the "Principle of Safe Substitution."

Lemma 1. *For any services $\sigma, \sigma' \in \Sigma$, if $\sigma \leq_\Sigma \sigma'$ then σ can safely be used in a context expecting service σ'.*

Proof. The proof follows by establishing the principle of safe substitution for objects, which is then used to prove the principle of safe substitution for services. Assume for sake of contradiction that for some objects x and y that $x \leq_\Theta y$ such that object x can not be safely used in a context expecting object y. Then under the set-theoretic interpretation of x and y as the sets s_x and s_y of all objects satisfying respective descriptions, s_x contains an element $o \notin s_y$. Since o is described by some $\alpha \in x$, and so by assumption that x is incompatible with y, it must be true that there is no $\beta \in y$ that describes x. However, this contradicts the definition of $\leq_\Theta$ that requires that there exists a conjunctive concept $\beta \in y$ that describes x. Therefore the principle of safe substitution holds for the lattice $(\Theta, \leq_\Theta)$.

Now assume that there exists a σ such that $\sigma \leq_\Sigma \sigma'$, but σ can not be used in a context expecting σ', where $\sigma = (A, I, O)$ and $\sigma' = (A', I', O')$. Clearly the context expects a service that can accept input of the type described by I', and since σ accepts a more general input type, every input of type I' is also an input of type I. This explains the contravariance of input in the definition of $\leq_\Sigma$. Now, since the expected service will output objects of type O', the context can handle a service that outputs objects of a more specific kind O by the principle of safe substitution for objects. Therefore σ can safely be used in place of σ'. □

Theorem 1. *(Soundness of Composables) For any services σ and σ', if service σ' can not be safely used in a context expecting service σ then $\sigma' \notin C(\sigma)$, that is the set of composables for a service does not contain any incompatible services.*

Proof. Assume the existence of services σ and σ' such that σ' can not be safely used in a context expecting service σ. By the principle of safe substitution it is not true that $\sigma' \leq_\Sigma \sigma$, and hence by the principle of extensionality, $\sigma' \notin C(\sigma)$. Therefore the set of composables only contains correct, i.e. compatible services. □

Theorem 2. *(Completeness of Composables) For any service σ, if there exists a service σ' that can safely be used in a context expecting service σ then $\sigma' \in C(\sigma)$, that is the set of composables contains all services compatible with a given description.*

Proof. Assume there exists such a service σ', then by the principle of safe substitution $\sigma' \leq_\Sigma \sigma$. So by the definition of $C(\sigma)$, $\sigma' \in C(\sigma)$, and therefore the set of composables is complete for arbitrary $\sigma \in \Sigma$. □

8 Comparison with OWL-S and WSML

OWL-S is another service description language [4], 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 underspecified. 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. Thus 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. The other recent approaches like WSMO,WSML etc also suffer from the same limitation. In contrast, USDL uses the universal WordNet ontology to take care of 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 complimentary. Thus USDL can be treated as an extension to OWL-S which makes OWL-S description easy to write and more complete.

Also, OWL-S can be regarded as the composition language for USDL. If a new service can be build 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. Then 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. 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. The USDL description also allows this composite service to be treated as an atomic service by some other application.

For example: The aforementioned *reserve* 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 *reserve* service by composing these three services in sequence using the OWL-S *Sequence* construct. The following is the OWL-S description of the composed *reserve* 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="reserve">
  <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 *reserve* service automatically by using the USDL descriptions of the component services for discovering them from the existing services. Once we have the component services, the OWL-S description can be used to generate the new composed service.

9 Other Related Work

Another related area of research involves message conversation constraints, also known as behavioral signatures [12, 13, 15]. Behavior signature models do not stray far from the explicit description of the lexical form of messages, they expect the messages to be lexically and semantically correct prior to verification via model checking. Hence behavior signatures deal with low-level functional implementation constraints, while USDL deals with higher-level real world concepts. However, USDL and behavioral signatures can be regarded as complimentary 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 [14].

10 Conclusion

In order to address the mounting complexity of information technology services integration, standards must be used so that services can be published and documented so that they can be reliably cataloged, searched, and composed in a semi-automatic to fully-automatic manner. 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 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 [8]. 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 [14]. Future work also includes developing tools that will allow automatic generation of new services based on combining USDL descriptions of existing atomic services. The interesting problem to be addressed is: can USDL description of such automatically generated services be also automatically generated?

References

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

[2] Resource description framework. `http://www.w3.org/RDF`.

[3] Sap interface repository. `http://ifr.sap.com/catalog/query.asp`.

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

[5] Web ontology language reference. `http://www.w3.org/TR/owl-ref`.

[6] Web services description language. `http://www.w3.org/TR/wsdl`.

[7] Wordnet: a lexical database for the english language. `http://www.cogsci.princeton.edu/~wn`.

[8] A. Bansal, K. Patel, and G. G. et al. Towards intelligent services: A case study in chemical emergency response. In *ICWS*, 2005.

[9] A. Bansal, L. Simon, A. Mallya, G. Gupta, and T. Hite. Automatic querying and composite services generation with usdl. Working paper, 2005.

[10] P. Bernstein. Generic model management - a database infrastructure for schema manipulation. In *CoopIS*, 2001.

[11] P. A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, pages 209–220, 2003.

[12] Z. Dang, O. H. Ibarra, and J. Sun. Composability of infinite-state activity automata. In *ISAAC*, 2004.

[13] C. E. Gerede, R. Hull, O. H. Ibarra, and J. Su. Automated composition of e-services:lookaheads. In *ICSOC*, 2004.

[14] T. Hite. Service composition and ranking: A strategic overview. Metallect Inc., 2005.

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

[16] S. Melnik, E. Rahm, and P.Bernstein. Rondo: A programming platform for generic model management. In *SIGMOD*, 2003.

[17] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.