OWL-S: Semantic Markup for Web Services

Tesseris George

(Postgraduate)

Baryannis George

(Postgraduate)

Computer Science Department

University of Crete

Table of Contents

Introduction

Web Services have enhanced the current Web by providing a new level of functionality. Nowadays, Web can be viewed not only as a distributed source of information, but also as a distributed source of services.

According to W3C, a Web service is a software system designed to support interoperable machine-to-machine interaction over a network and it has an interface described in a machine-processable format (specifically WSDL). Web Services Description Language (WSDL) specifies a protocol and encoding independent mechanism for Web Service providers. It is an XML vocabulary which describes network-reachable services and maps these to a messaging-capable collection of communication endpoints. Although it is capable to describe the means of interacting with offered services, it does not contain the expressiveness needed to describe the web service capabilities and requirements in an unambiguous and machine understandable fashion. Current research efforts aim to provide semantic descriptions of web services by using conceptualized knowledge, called ontology. An ontology is a vocabulary for describing a set of concepts within a domain (a domain is defined as a specific subject area or area of knowledge) and the relationships that exist between those concepts. It is used to reason about the properties of that domain, and may be used to define the domain. In the context of Web Services, ontologies figure prominently as a way of providing semantic descriptions for Web Services which can be used by web applications and intelligent agents.

The augmentation of Web service descriptions leads to what it is called Semantic Web Services. The semantic description of Web Services in an unambiguous and machine-understandable manner will have a great impact in areas such as e-Business, and Enterprise Application Integration, as it can enable dynamic, scalable and cost effective collaboration between different systems and organizations. These great potentials have made Semantic Web Services nowadays one of the most relevant research topics.

The outline of this report is as follows. Section 1 presents the limitations in the current standard models used to describe Web Services. The motivation behind Semantic Web Services is described in Section 2. Section 3 is a complete and thorough analysis of the

OWL-S ontologies while the following Section depicts the advanced features of OWL (in comparison with RDFS) that make it a suitable ontology language for semantic Web Service description. Finally, Section 5 concludes.

1. Syntactic Description Limitations

WSDL describe web service capabilities by defining parameters and operations that the service supports and associating parameters with abstract data types. Consider a currency converter service from US dollars to Euros named ConvertDollarsToEuros. The service specifies one input parameter named dollars of type float and one output parameter named euros of the same type (Table 1-1). To invoke the service the amount on money in dollars given and the converted amount of money is returned.

ConvertDollarsToEuros		
	Type	**Parameter Name**
Input	Float	Dollars
Output	Float	Euros

Table 1-1 Currency Converter Example

The problem with this description is that agents, which are programs that act on the behalf of their owner (human or other program), can not deduct what the service does. Agents see only parameter names and cannot, deduct their meaning by reading the names, as humans can. The only thing that agents can infer is that the parameters are of type float. But what does a service do if it requests a float and outputs a float? Two services can have the same syntactic description but perform completely different functions. Similarly, two syntactically different services can compute the same function. For example, one service can expose the individual parameters separately, while another one packages them in an XML document. Syntactically these two services are very different but it is possible to compute the same function. WSDL describes only the functional and syntactic aspects of a service. It does not provide behavioral or non-functional information of services. This has as result tasks such as discovery, invocation, composition and interoperation of web services not to be automated because a computer-interpretable description of the service is needed.

In order to address the above issue, research community has proposed Semantic markup of Web Services using ontologies. The following section describes in detail the tasks, which ontologies aim to enable.

2. Motivation behind ontology languages

Ontologies, such as OWL-S, in order to lead web services to their full potential, aim to enable primarily three tasks, automatic web service discovery, automatic web service invocation, automatic web service composition and interoperation.

Automatic web service discovery is an automated process for locating web services that provide a particular functionality and that adhere to requested properties. To provide such an automatic location, the discovery process needs not only a matching algorithm to match the respective descriptions, but also a language to declarative expresses the capabilities of services. For example, if a user wants to find a service that sells air tickets (service capability) between two given cities and accepts a specific credit card (both constrains) the task must be performed manually using a search engine and then determine if the service found satisfies the constrains.

Automatic web service invocation is the autonomous execution of a web service, given that a corresponding web service has been found, without any human interaction. For example, if the execution of the service includes multiple steps, the agent needs to know how to interact with the service in order to complete the required sequence. Invocation information presented by a given service must be agnostic in principle with respect to the specific technologies which will ground it. Nevertheless, details must be available at run-time for the service requester in order to perform a real invocation.

Automatic web service composition and interoperation is the combination and interoperation of several web services to fulfill a certain objective. For example, a travel agency wants to provide a full travel package to its customers. This Service might be composed of several individual Web Services, such as Book a Hotel, Book a Flight, Book a Car etc. In the following sections OWL-S is presented.

3. OWL-S

OWL-S (previously called DAML-S) stands for Web Ontology Language for Services and it is an OWL ontology/language to formally describe Web services. It comprise three main subontologies as illustrated in figure 3-1, known as the service profile, service process model, and service grounding.

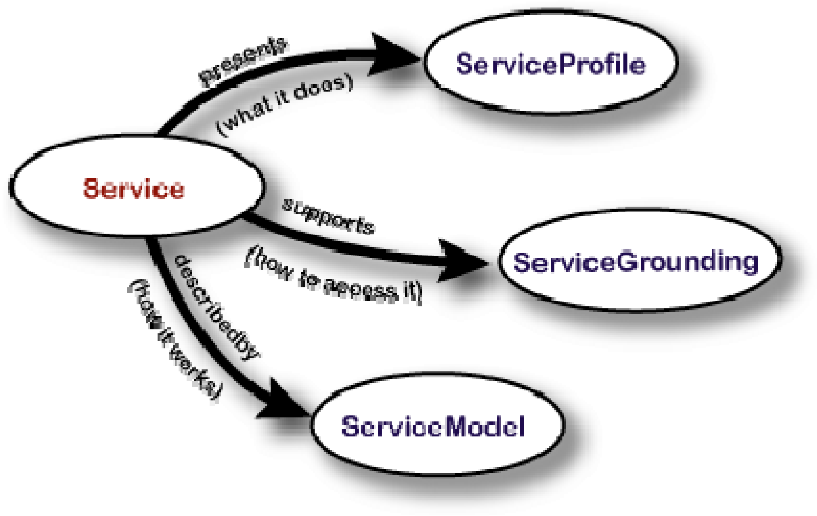


Figure 3-1 OWL-S Subontologies

This structuring of the ontology aims to provide three essential types of knowledge about a service. The service profile subontology is used to describe what the service provides for the prospective clients. This information is used to advertise the service, construct service requests and perform matchmaking. The service process model describes how a service works in order to enable invocation, enactment, composition, monitoring and recovery. Finally, the service grounding specifies how to access the service by providing the needed details about transport protocols. Service Profile, Service Grounding and Service model are described in detail in the following three subsections.

3.1 Service Profile

The first step in an interaction involving web services is to discover those relevant web services that match perfectly or partially to the needs and requirements of the interaction. To that end, service providers need to advertise the services they offer, in such a way that the service requesters can easily find what they are looking for. Furthermore, in the Semantic web, these advertisements must be semantically annotated with machine-interpretable metadata so that agents can automatically reason about this metadata and find matching web services. The Service Profile class in the OWL-S ontology aims to do exactly that.

The OWL-S ontologies allow for a service provider to describe the web services it offers by creating a customized subclass of the main Service Profile class. This subclass can contain an arbitrary amount of information while domain ontologies can be created to describe related web services. The OWL-S specification provides the Profile class which is one possible representation of a web service, but is neither mandatory nor restrictive and service providers are free to adapt the Profile class as necessary or create completely different profile classes.

The Profile class

The Profile subclass describes three dimensions of a web service: the service provider, the service functionality and a set of service characteristics. The first dimension deals with the entity that provides the service and contains contact information to anyone that may be associated with the service, such as the people responsible for running and maintaining the service instances or people responsible for informing the service requesters in detail.

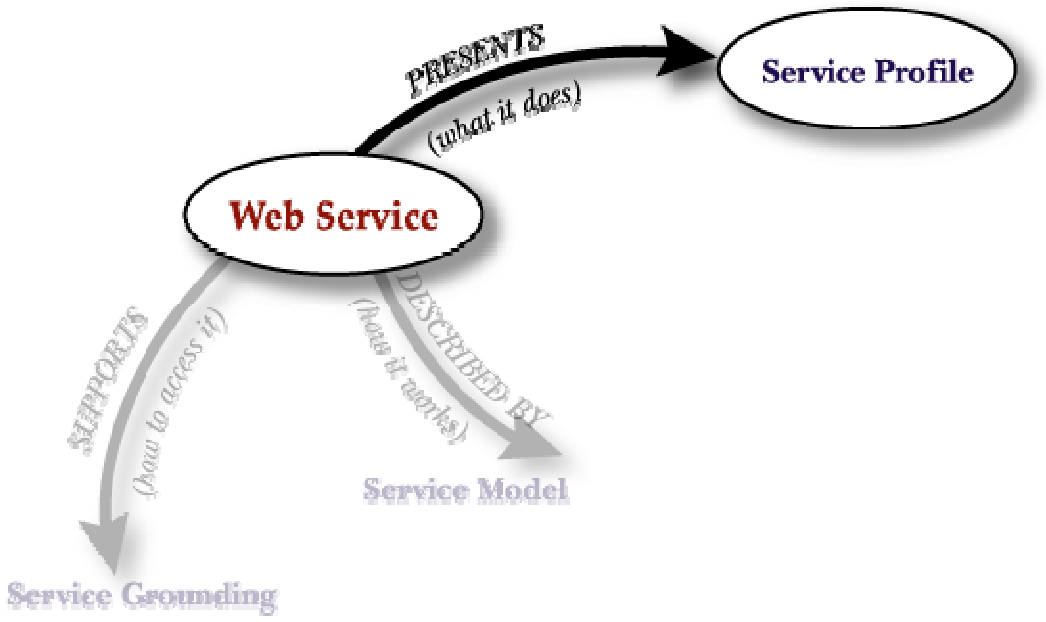
The second dimension is essentially the functional description of the web service. It contains the inputs that are necessary for the service to be executed, the preconditions that must be met to ensure a valid execution, the information that is generated as the output of the service as well as any effects to the state of the world that result from the execution of the service. This set of information is referred to as IOPEs (Inputs, Outputs, Preconditions and Effects). In some cases, it is interesting to couple an output and an effect as they are directly related and as a result the functional description is also referred to as IOPRs (Inputs, Outputs, Preconditions and Result, where a Result is a coupled output and effect). In the description of the Service Model class in the next section, IOPRs will be revisited as the functional description is part of that class as well. The relation between a service profile and a service model with regard to the functional description will be further explored in that section.

The third dimension of the information contained in the Profile class essentially covers all other features that one can see fit to include in a service description. OWL-S version 1.1 goes into some level of detail on what features may be included but version 1.2 which is still in prerelease essentially leaves this part of the Profile class for the user to specify, as explained in a later subsection. According to version 1.1 specification, a Profile may contain information about the category of the service using an existing classification system such as the United Nations Standard Products and Services Code (UNSPSC). Also, a very important feature that should be part of a service description is the Quality of Service (QoS). QoS is a major factor in service discovery and selection, as searching only based on the functional description will yield services that may advertise the required operations, but may also be unreliable, slow or even malicious. Finally, it is at the liberty of the service providers to include any other parameter to describe their services, from an estimate of the max response time to the geographic availability of the service, to cost-related parameters etc.

The Service Profile superclass

All profile ontologies that one can create to describe a service using OWL-S are subclasses of the Service Profile class, including the Profile class described above. The

Service Profile superclass and thus all profile subclasses instances are linked with a service instance as shown in the following figure:



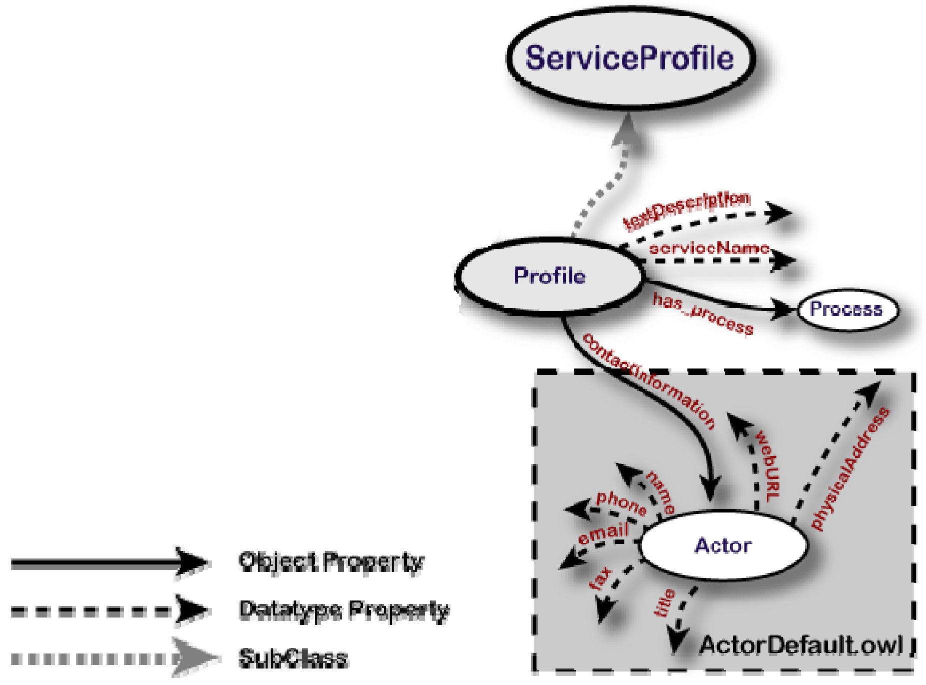
The relation “presents” links an instance of a web service with an instance of an associated profile, stating the equivalent of the natural language phrase “this service is described by this profile”. To describe the opposite relation, “this profile describes this service”, the inverse relation of “presents”, “presentedBy” is used. Thus, a two-way relation between a service and a profile is established, relating a service to a profile and a profile to a service. It should be noted that no cardinality constraints are declared for these relations, meaning that a service may be linked to an arbitrary number of profiles (or even none) and a profile may describe more than one services or even no service. This allows for multiple profiles which may be useful when providing a service that may be addressed to different groups of consumers. It also enables partial characterization of a service, i.e. describing services using only a service model and a service grounding instance.

Following is part of the Service.owl ontology file that declares the above relations

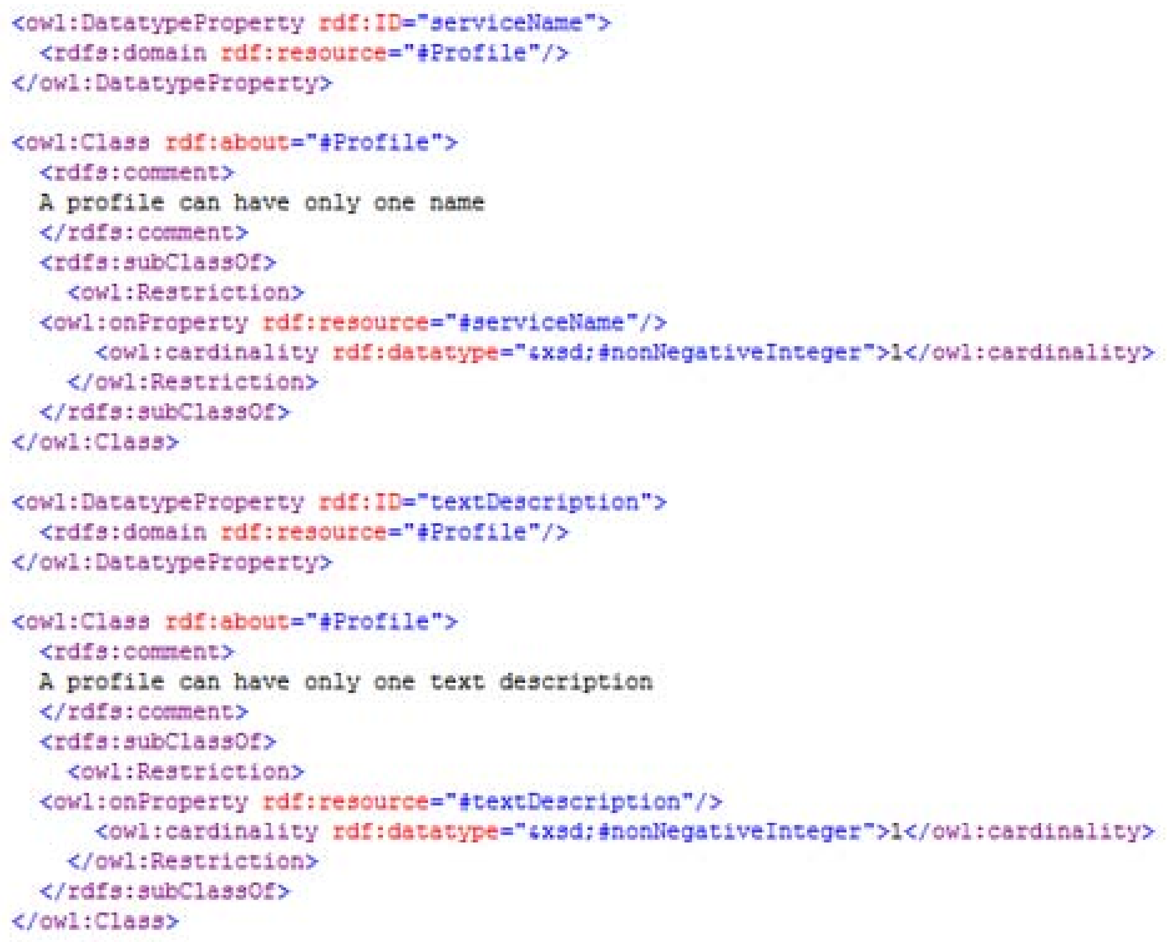
```
<!--  Presenting a profile    -->

<owl:ObjectProperty rdf:ID="presents">
  <rdfs:comment>
    There are no cardinality restrictions on this property.
  </rdfs:comment>
  <rdfs:domain rdf:resource="&service;#Service"/>
  <rdfs:range rdf:resource="&service;#ServiceProfile"/>
  <owl:inverseOf rdf:resource="&service;#presentedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="presentedBy">
  <rdfs:comment>
    There are no cardinality restrictions on this property.
  </rdfs:comment>
  <rdfs:domain rdf:resource="&service;#ServiceProfile"/>
  <rdfs:range rdf:resource="&service;#Service"/>
  <owl:inverseOf rdf:resource="&service;#presents"/>
</owl:ObjectProperty>
```

Basic Description and Contact Information

The first batch of profile properties gives basic information for the service and its provider.



The serviceName property refers to the name of the service that is described. This name may be used as an identifier for the service, provided that it is unique. The textDescription provides a brief textual description of the service, including its functionality, requirements and any other information the provider wants to include. These first two properties contain human-readable and agents are most likely unable to automatically interpret and process them. Also, a profile can have at most one serviceName and textDescription. The corresponding code is shown below:



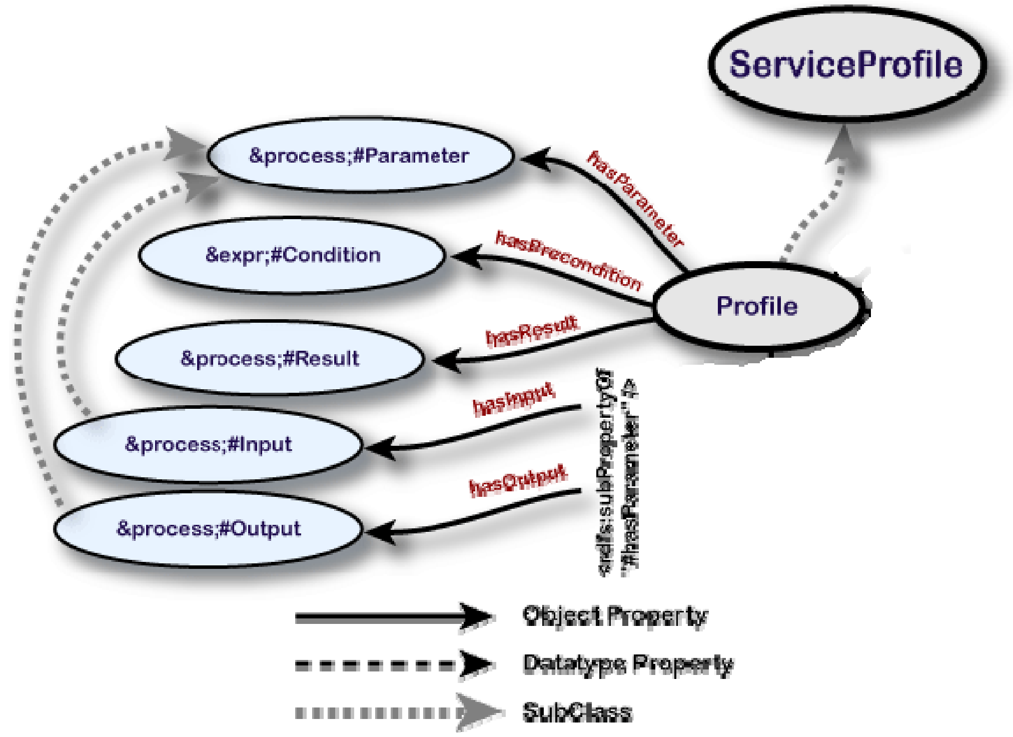
```
<owl:DatatypeProperty rdf:ID="serviceName">
  <rdfs:domain rdf:resource="#Profile"/>
</owl:DatatypeProperty>

<owl:Class rdf:about="#Profile">
  <rdfs:comment>
  A profile can have only one name
  </rdfs:comment>
  <rdfs:subClassOf>
    <owl:Restriction>
  <owl:onProperty rdf:resource="#serviceName"/>
      <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:DatatypeProperty rdf:ID="textDescription">
  <rdfs:domain rdf:resource="#Profile"/>
</owl:DatatypeProperty>

<owl:Class rdf:about="#Profile">
  <rdfs:comment>
  A profile can have only one text description
  </rdfs:comment>
  <rdfs:subClassOf>
    <owl:Restriction>
  <owl:onProperty rdf:resource="#textDescription"/>
      <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

As far as describing contact information is concerned, the contactInformation property is provided. Its range is not limited, as OWL-S allows for any ontology to be used such as FOAF or VCard and also provides a simple Actor class as an alternative. This class is declared in a separate file and its basic properties are shown in the figure above. These include the name and title of the entity, the phone, fax, email and physical address as well as a URL address. The range of all these properties is xsd:string. Finally, the has_process property links the Profile class with an instance of the Process class, as declared in the Service Model ontology which will be described in a following section.

Functional Description

The functional description of a web service is an essential part of its profile, since it deals with its functionality, i.e. what operations it provides to the requester. The functional description has two aspects: the first involves the information transformation performed by the service and is represented by inputs and outputs while the second deals with the change to the state of the world caused by the execution of the service and is represented by the preconditions and effects. For example, if we want to describe a book-selling service, we need to describe its inputs (book ID, credit card number, credit card expiration date etc.) and its preconditions (validity of credit card, balance higher than zero). Also, the output of the service needs to be described (a transaction receipt) and an effect (money moved from buyer's account to seller's and book sent to the buyer from the seller warehouse).

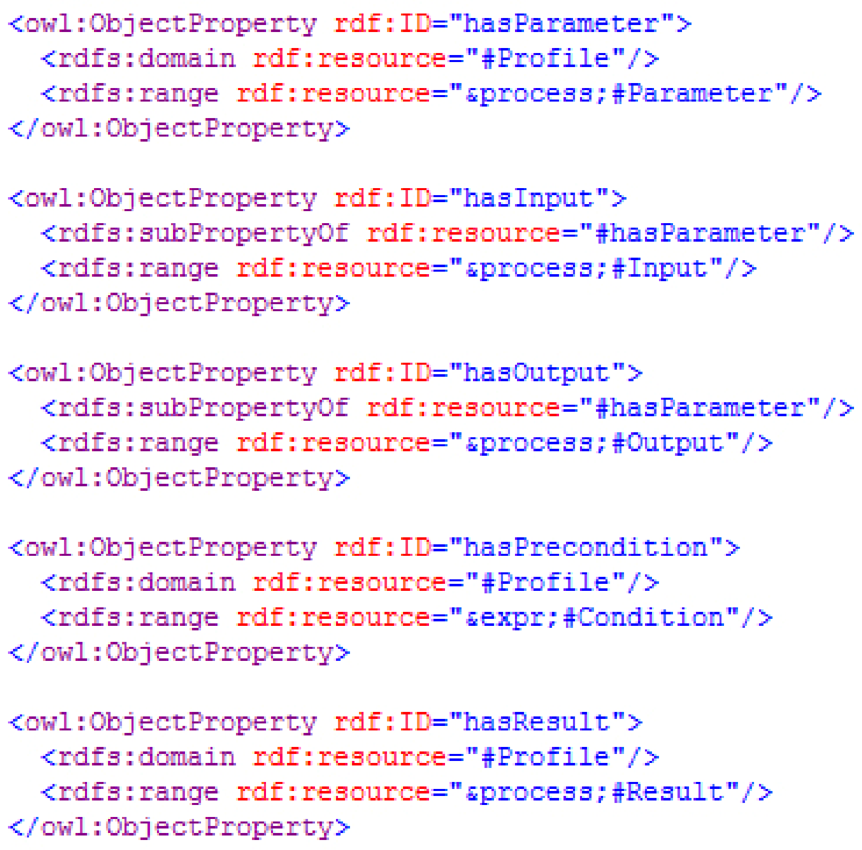
In the following figure, the functional properties of a service profile are illustrated:



The hasParameter property groups inputs and outputs together, by providing a superclass which they inherit. This property is not instantiated as all inputs and outputs are instances of the subclasses of hasParameter, hasInput and hasOutput. This grouping signifies the conceptual similarity between inputs and outputs as they are both involved in information transformation in contrast with preconditions and effects that deal with state change.

The hasInput and hasOutput properties are used to describe inputs and outputs of the service respectively. The hasPrecondition property describes service preconditions. Finally, the hasResult describes a coupled effect and output. It specifies both the conditions under which the outputs are generated and the domain changes that are caused by the service execution.

It should be noted that no schema for these parameters is provided by the Profile ontology. Such a schema exists only in the Process ontology, a subclass of the Service Model class which is described in the next section. This is due to the close relation between a service profile and a service model with regard to the functional description and will be further explored in that section. As a result, for each IOPR, a single instance is created in the Process ontology of the Service Model while the instance placed in the Profile is simply a pointer to that single instance. This, however, doesn't disallow the Profile from creating its own IOPR instances using the schema provided by the Process ontology. The code corresponding to the functional description is shown below:



```
<owl:ObjectProperty rdf:ID="hasParameter">
  <rdfs:domain rdf:resource="#Profile"/>
  <rdfs:range rdf:resource="&process;#Parameter"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasInput">
  <rdfs:subPropertyOf rdf:resource="#hasParameter"/>
  <rdfs:range rdf:resource="&process;#Input"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasOutput">
  <rdfs:subPropertyOf rdf:resource="#hasParameter"/>
  <rdfs:range rdf:resource="&process;#Output"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasPrecondition">
  <rdfs:domain rdf:resource="#Profile"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasResult">
  <rdfs:domain rdf:resource="#Profile"/>
  <rdfs:range rdf:resource="&process;#Result"/>
</owl:ObjectProperty>
```

Other Profile Parameters

The parameters in this subsection are declared in OWL-S version 1.1 but will be deprecated from version 1.2 onwards. The serviceCategory property links a Profile with a class that describes the category to which a service belongs. The category classification may be outside OWL-S or even outside OWL. The ServiceCategory contains a series of properties that identify the category instance. These properties are:

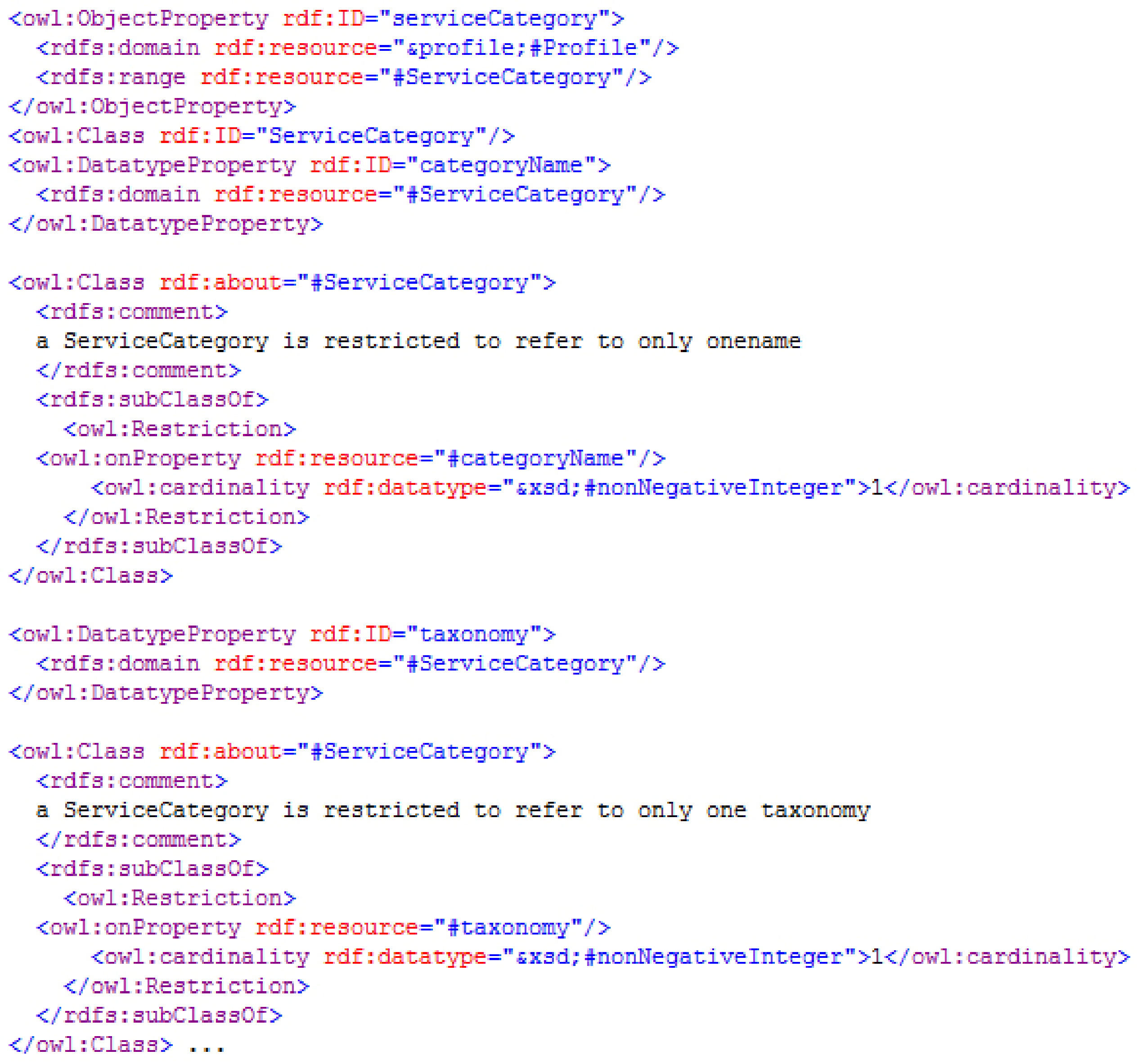
categoryName: the name of the service category. It can be a literal or a URI.

taxonomy: a reference to the taxonomy scheme of the category, that can be either a URI of the taxonomy, the URL where the taxonomy can be found or the name of the taxonomy or anything else.

value: the value in the above taxonomy. More than one value properties can be linked with a single ServiceCategory instance.

code: the code associated to the taxonomy for each service type.

Part of the code related to the ServiceCategory properties is shown below:



```
<owl:ObjectProperty rdf:ID="serviceCategory">
  <rdfs:domain rdf:resource="&profile;#Profile"/>
  <rdfs:range rdf:resource="#ServiceCategory"/>
</owl:ObjectProperty>
<owl:Class rdf:ID="ServiceCategory"/>
<owl:DatatypeProperty rdf:ID="categoryName">
  <rdfs:domain rdf:resource="#ServiceCategory"/>
</owl:DatatypeProperty>

<owl:Class rdf:about="#ServiceCategory">
  <rdfs:comment>
  a ServiceCategory is restricted to refer to only onename
  </rdfs:comment>
  <rdfs:subClassOf>
    <owl:Restriction>
  <owl:onProperty rdf:resource="#categoryName"/>
      <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:DatatypeProperty rdf:ID="taxonomy">
  <rdfs:domain rdf:resource="#ServiceCategory"/>
</owl:DatatypeProperty>

<owl:Class rdf:about="#ServiceCategory">
  <rdfs:comment>
  a ServiceCategory is restricted to refer to only one taxonomy
  </rdfs:comment>
  <rdfs:subClassOf>
    <owl:Restriction>
  <owl:onProperty rdf:resource="#taxonomy"/>
      <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class> ...
```

The OWL-S specification also declares two other grouping-related properties for the Profile class, serviceClassification and serviceProduct. The serviceClasification property defines a mapping from a Profile instance to an OWL ontology of service classification such as an OWL specification of NAICS (North American Industry Classification System). On the other hand, the serviceProduct property defines a mapping from a Profile instance to an OWL ontology of products, such as an OWL specification of UNSPSC (United Nations Products and Services Code). The difference between these properties and the serviceCategory class is that while both aim to place the service in an existing classification system, the serviceClassification and serviceProduct have values that are OWL instances which is not the case for serviceCategory instances which are simple strings referring to any taxonomy OWL or not. The code related to these properties is shown below:

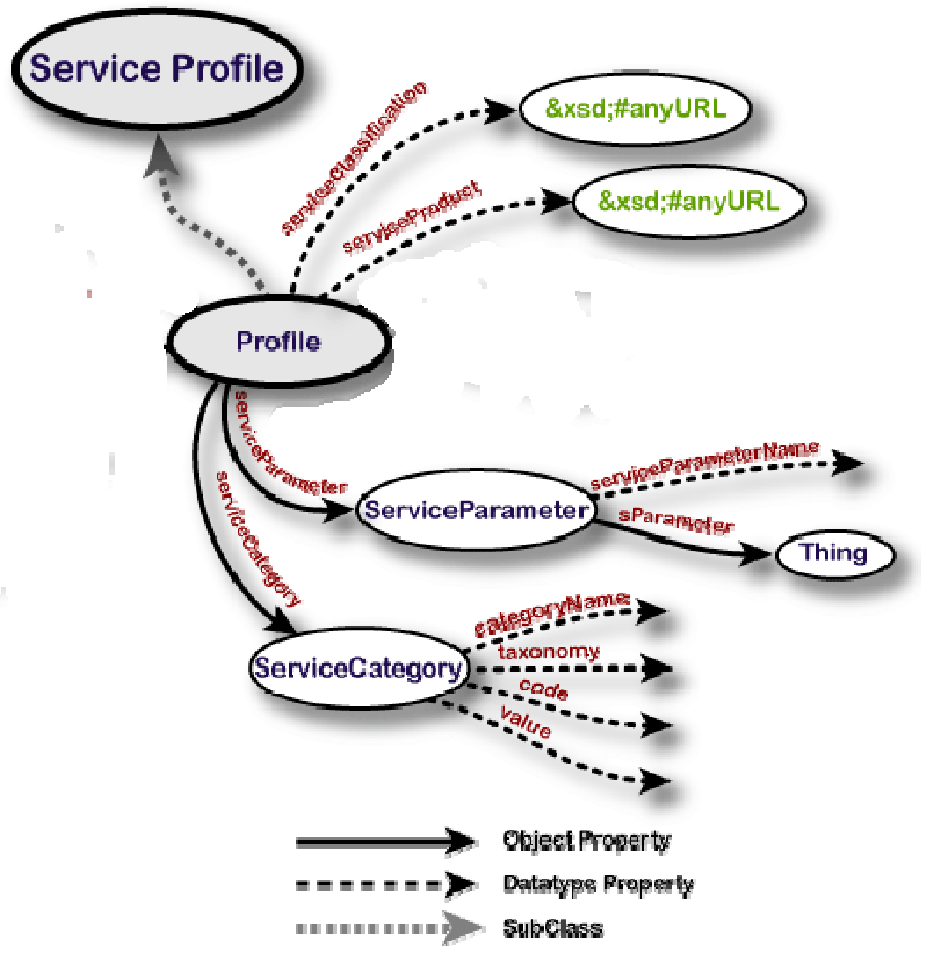
```
<owl:ObjectProperty rdf:ID="serviceCategory">
  <rdfs:domain rdf:resource="#Profile"/>
  <rdfs:range rdf:resource="#ServiceCategory"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="serviceParameter">
  <rdfs:domain rdf:resource="#Profile"/>
  <rdfs:range rdf:resource="#ServiceParameter"/>
</owl:ObjectProperty>
```

Finally, OWL-S offers a construct to declare any other property that may be included in a service profile. A ServiceParameter class is linked to the Profile class using the serviceParameter property. Furthermore, the ServiceParameter has two properties, serviceParameterName and sParameter. The serviceParameterName is the name of the parameter, either a literal or the URI of the parameter while SParameter points to the value of the parameter within some OWL ontology. The code related to these properties is shown below:

```
<owl:Class rdf:ID="ServiceParameter"/>

<owl:DatatypeProperty rdf:ID="serviceParameterName">
  <rdfs:domain rdf:resource="#ServiceParameter"/>
</owl:DatatypeProperty>

<owl:Class rdf:about="#ServiceParameter">
  <rdfs:comment>
  A ServiceParameter should have at most 1 name (more precisely only
    one serviceParameterName)
  </rdfs:comment>
  <rdfs:subClassOf>
    <owl:Restriction>
  <owl:onProperty rdf:resource="#serviceParameterName"/>
      <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

```
<owl:ObjectProperty rdf:ID="sParameter">
  <rdfs:domain rdf:resource="#ServiceParameter"/>
  <rdfs:range  rdf:resource="&owl;#Thing"/>
</owl:ObjectProperty>

<owl:Class rdf:about="#ServiceParameter">
  <rdfs:comment>
  a Parameter is restricted to refer to only one concept in some
  ontology
  </rdfs:comment>
  <rdfs:subClassOf>
    <owl:Restriction>
  <owl:onProperty rdf:resource="#sParameter"/>
      <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

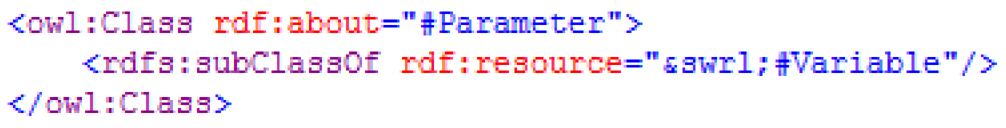
The parameters presented in this subsection are illustrated in the following figure:



3.2 Service Model

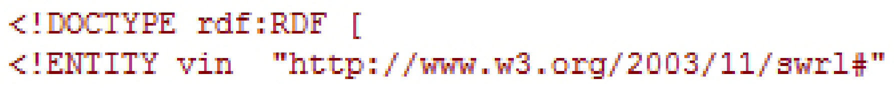
The Process model in OWL-S describes the service functionality and specifies the ways a client may interact with the service in order to achieve its functionality. This is done by expressing the data transformation with the inputs and the outputs and the state transformation with the preconditions and effects. Although the Profile and the Process Model play different roles during Web Service lifecycle they are two different representations of the same service, so it is natural to expect that the input, output, precondition, and effects (IOPEs) of one are reflected in the IOPEs of the other. There are no constraints between Profiles and Process Models descriptions, so they may be inconsistent without affecting the validity of the OWL expression. Still, if the Profile represents a service that is not consistent with the service represented in the Process Model, the interaction will break at some point. Although the Profile of a service provides a concise description of the service to a registry, once the service has been selected the Profile is useless and the client will use the Process Model to control the interaction with the service. It is important to understand that a process is not a program to be executed, but a specification of the ways a client may interact with a service.

Inputs and outputs are subclasses of a general class called Parameter. It's convenient to identify parameters with what are called variables in Semantic Web Rule Language (SWRL), the language for expressing OWL Rules. So, every parameter is subclass of swrl Variable.



```
<owl:Class rdf:about="#Parameter">
    <rdfs:subClassOf rdf:resource="&swrl;#Variable"/>
</owl:Class>
```

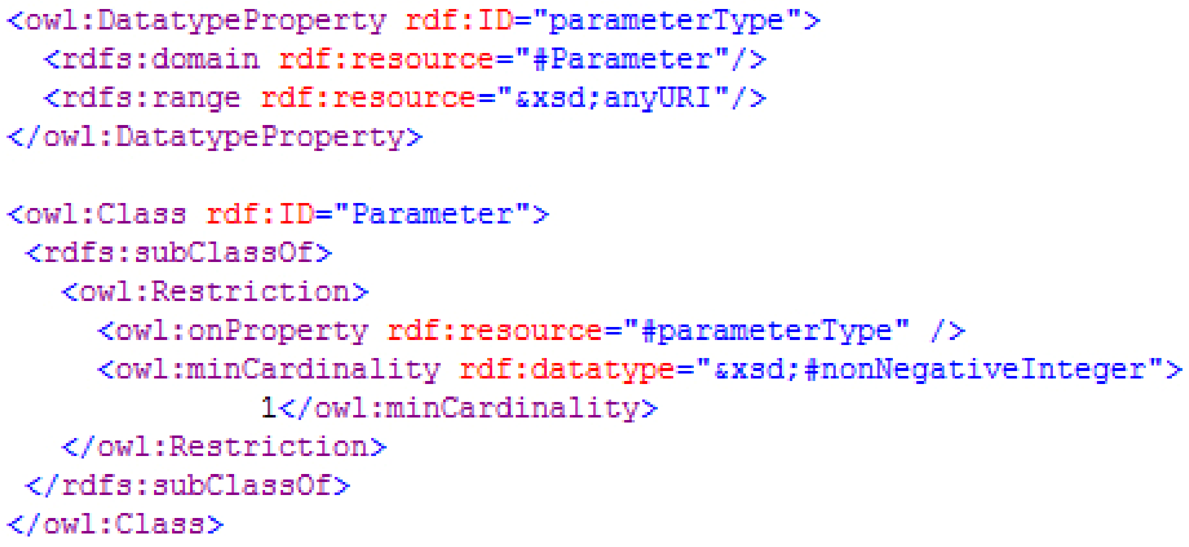
Obviously &swrl; has to be declared previously.



```
<!DOCTYPE rdf:RDF [
<!ENTITY vin  "http://www.w3.org/2003/11/swrl#"
```

After this ENTITY declaration, we could write the value "&swrl;#Variable" and it would expand to "http://www.w3.org/2003/11/swrl#Variable".

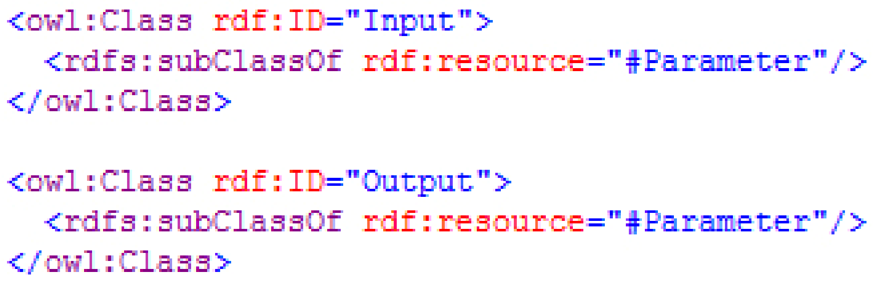
Every parameter has a type (parameterType), specified using a URI. This is not the OWL class the parameter belongs to, but a specification of the class (or datatype) that values of the parameter belong to. rdfs:domain is used to state that any resource that has a given property is an instance of one or more classes and rdfs:range is used to state that the values of a property are instances of one or more classes.



```
<owl:DatatypeProperty rdf:ID="parameterType">
  <rdfs:domain rdf:resource="#Parameter"/>
  <rdfs:range rdf:resource="&xsd;anyURI"/>
</owl:DatatypeProperty>

<owl:Class rdf:ID="Parameter">
 <rdfs:subClassOf>
   <owl:Restriction>
     <owl:onProperty rdf:resource="#parameterType" />
     <owl:minCardinality rdf:datatype="&xsd;#nonNegativeInteger">
              1</owl:minCardinality>
   </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

The owl:restriction element from the above code defines an unnamed class (which is called anonymous class) that represents the set of things with at least one parameterType property. Including this restriction in the Parameter class definition body we state that parameters are also members of this anonymous class. Thus, every individual parameter must participate in at least one parameterType relation.

As mentioned before, inputs and outputs specify the data transformation produced by the process. They are subclasses of parameter:



```
<owl:Class rdf:ID="Input">
  <rdfs:subClassOf rdf:resource="#Parameter"/>
</owl:Class>

<owl:Class rdf:ID="Output">
  <rdfs:subClassOf rdf:resource="#Parameter"/>
</owl:Class>
```

hasParameter property has the subproperties hasInput, hasOutput, and hasLocal. It ranges over a Parameter instance of the Process ontology:

```
<owl:ObjectProperty rdf:ID="hasParameter">
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Parameter"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasInput">
  <rdfs:subPropertyOf rdf:resource="#hasParameter"/>
  <rdfs:range rdf:resource="#Input"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasOutput">
  <rdfs:subPropertyOf rdf:resource="#hasParameter"/>
  <rdfs:range rdf:resource="#Output"/>
</owl:ObjectProperty>
```

Local variables are an advanced feature of OWL-S (added in version 1.1).They are variables to be bound in preconditions and then used to specify result conditions, outputs and effects.

```
<owl:ObjectProperty rdf:ID="hasLocal">
  <rdfs:subPropertyOf rdf:resource="#hasParameter"/>
  <rdfs:range rdf:resource="#Local"/>
</owl:ObjectProperty>
```

Preconditions determine if a process can be performed successfully. Unless the precondition is true the process cannot be performed successfully.

```
<owl:ObjectProperty rdf:ID="hasPrecondition">
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>
```

A process may change the state of the world (effects) and lead to different information transformation (outputs), depending on the context of execution. The Result class is used to couple outputs and effects and is related to Process by hasResult. Thus, multiple couples of outputs and effects (results) may be associated with a process.

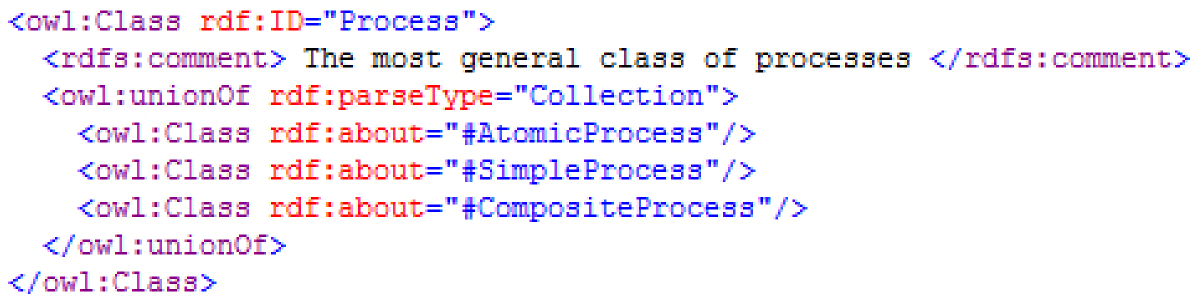
```
<owl:Class rdf:ID="Result">
  <rdfs:label>Result</rdfs:label>
</owl:Class>

<owl:ObjectProperty rdf:ID="hasResult">
  <rdfs:label>hasResult</rdfs:label>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Result"/>
</owl:ObjectProperty>
```

The process model then can describe the result as follows:

```
<owl:ObjectProperty rdf:ID="inCondition">
  <rdfs:label>inCondition</rdfs:label>
  <rdfs:domain rdf:resource="#Result"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasResultVar">
  <rdfs:label>hasResultVar</rdfs:label>
  <rdfs:domain rdf:resource="#Result"/>
  <rdfs:range rdf:resource="#ResultVar"/>
</owl:ObjectProperty>


<owl:ObjectProperty rdf:ID="withOutput">
  <rdfs:label>withOutput</rdfs:label>
  <rdfs:domain rdf:resource="#Result"/>
  <rdfs:range rdf:resource="#OutputBinding"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasEffect">
  <rdfs:label>hasEffect</rdfs:label>
  <rdfs:domain rdf:resource="#Result"/>
  <rdfs:range rdf:resource="&expr;#Expression"/>
</owl:ObjectProperty>
```

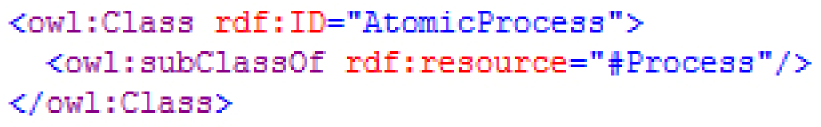
Four properties are used, inCondition, hasResultVar, withOutput and hasEffect. If there is no inCondition property then the result condition is TRUE and always occurs when the process is executed. If multiple inCondition properties exist then they are conjoined, so they must all be true for the result to occur (that is, for the outputs and effects to be realized). Outputs and effects are associated with a result using the withOutput and hasEffect properties correspondingly. Finally, ResultVar is used to declare an (implicitly universally quantified) variable that is referenced in the result condition but is not a process parameter.

OWL-S differentiates between three types of processes atomic processes, composite processes and simple processes:



```
<owl:Class rdf:ID="Process">
  <rdfs:comment> The most general class of processes </rdfs:comment>
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#AtomicProcess"/>
    <owl:Class rdf:about="#SimpleProcess"/>
    <owl:Class rdf:about="#CompositeProcess"/>
  </owl:unionOf>
</owl:Class>
```

An atomic process is a description of a service that expects one (possibly complex) message and returns one (possibly complex) message in response. It is the basic unit of implementation and directly invokable by passing the appropriate parameters and execute in a single step. An atomic process is a "black box" representation; that is, no description is given of how the process works (apart from inputs, outputs, preconditions, and effects).



```
<owl:Class rdf:ID="AtomicProcess">
  <owl:subClassOf rdf:resource="#Process"/>
</owl:Class>
```

A simple process is not directly invokable and is executed in a single step. It provides, through an abstraction mechanism, different views of atomic processes (specialized ways of using) or simplifed representations of composite processes (using the "realizedBy" and expandsTo" properties correspondingly) for purposes of planning and reasoning.

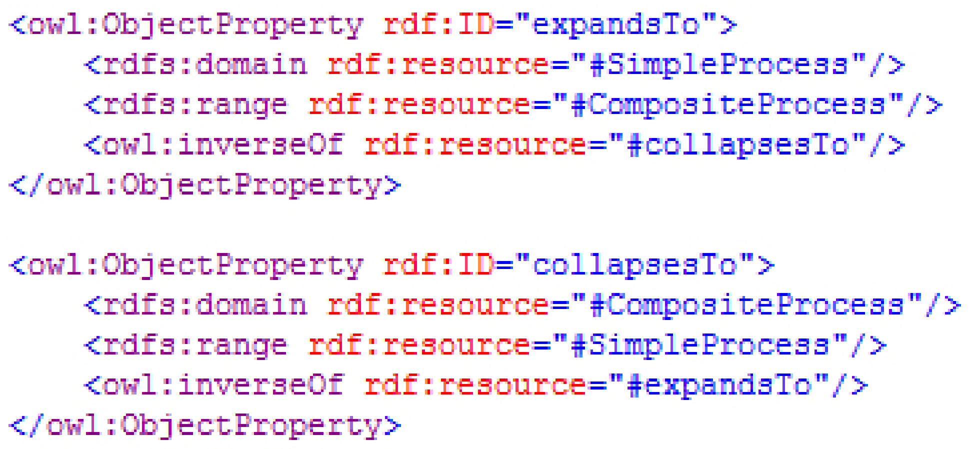
disjointWith guarantees that a SimpleProcess that is a member of Process cannot simultaneously be an instance of an AtomicProcess.

```
<owl:Class rdf:ID="SimpleProcess">
  <rdfs:subClassOf rdf:resource="#Process"/>
  <owl:disjointWith rdf:resource="#AtomicProcess"/>
</owl:Class>
```

In the following code, inverseOf states that realizedBy property is the inverse of realizes property and vice versa. Therefore, a reasoner knowing that a Simple Process is realized by an Atomic Process can deduce that the Atomic Process realizes the Simple Process and vice versa.

```
<owl:ObjectProperty rdf:ID="realizedBy">
  <rdfs:domain rdf:resource="#SimpleProcess"/>
  <rdfs:range rdf:resource="#AtomicProcess"/>
  <owl:inverseOf rdf:resource="#realizes"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="realizes">
  <rdfs:domain rdf:resource="#AtomicProcess"/>
  <rdfs:range rdf:resource="#SimpleProcess"/>
  <owl:inverseOf rdf:resource="#realizedBy"/>
</owl:ObjectProperty>
```

There are two fundamental relations that can hold between simple processes and composite processes. The first refers to "expanding" a process to its underlying CompositeProcess (zoom-in) while the other corresponds to "collapsing" a composite process into a simple process (zoom-out). Same logic with realizedBy property is applied here:



```
<owl:ObjectProperty rdf:ID="expandsTo">
    <rdfs:domain rdf:resource="#SimpleProcess"/>
    <rdfs:range rdf:resource="#CompositeProcess"/>
    <owl:inverseOf rdf:resource="#collapsesTo"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="collapsesTo">
    <rdfs:domain rdf:resource="#CompositeProcess"/>
    <rdfs:range rdf:resource="#SimpleProcess"/>
    <owl:inverseOf rdf:resource="#expandsTo"/>
</owl:ObjectProperty>
```

A composite process is one that maintains some state; each message the client sends, advances through the process. The composite process can consist of sub-processes. As mentined before, process can have two sorts of purpose. First, it can generate and return some new information based on information it is given and the world state. Information production is described by the inputs and outputs of the process. Second, after a successful execution it can produce a change in the world. This transition is described by the preconditions and effects of the process. A process can have any number of inputs (including zero), representing the information that is, under some conditions, required for the performance of the process. It can have any number of outputs, the information that the process provides to the requester. There can be any number of preconditions, which must all hold in order for the process to be successfully invoked. Finally, the process can have any number of effects. Outputs and effects can depend on conditions that hold true of the world state at the time the process is performed.

```
<owl:Class rdf:ID="CompositeProcess">
  <rdfs:subClassOf rdf:resource="#Process"/>
  <owl:disjointWith rdf:resource="#AtomicProcess"/>
  <owl:disjointWith rdf:resource="#SimpleProcess"/>
  <rdfs:comment>
    A CompositeProcess must have exactly 1 composedOf property.
  </rdfs:comment>
  <owl:intersectionOf rdf:parseType="Collection">
      <owl:Class rdf:about="#Process"/>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#composedOf"/>
         <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">
                  1</owl:cardinality>
      </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

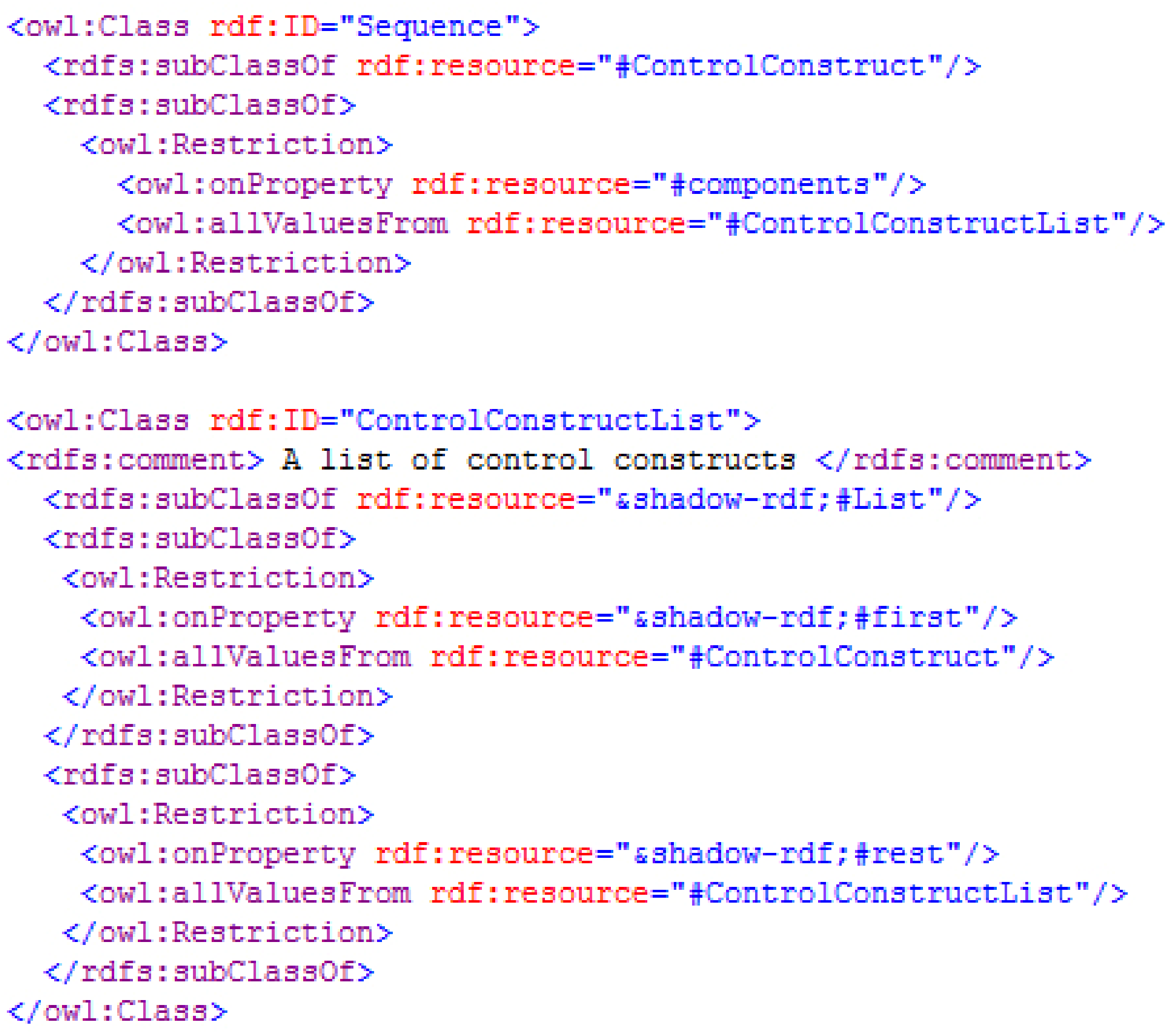
disjointWith guarantees that a CompositeProcess that is a member of Process cannot simultaneously be an instance of an AtomicProcess or SimpleProcess. A composite process is composed of subprocesses, and specifies constraints on the ordering and conditional execution of these subprocesses. These constraints are captured by the composeOf property, thus is a required property for every CompositeProcess.

```
<owl:ObjectProperty rdf:ID="composedOf">
  <rdfs:domain rdf:resource="#CompositeProcess"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="ControlConstruct">
</owl:Class>
```

composeOf property uses an additional Property, called components. This property represents the building elements of composeOf property which are nested control constructs, and, in some cases, defines their ordering.

```
<owl:ObjectProperty rdf:ID="components">
    <rdfs:domain rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>
```

Composite processes references to processes (Atomic or Composite) called PERFORMs. Perform is a control construct specifying where the client should invoke a process provided by some server. control constructs can be Sequence, Split, Split + Join, Any-Order, Condition, If-Then-Else,Iterate, Repeat-While and Repeat-Until. For example, sequence is defined as having a list of component processes that specify the body:



```
<owl:Class rdf:ID="Sequence">
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#components"/>
      <owl:allValuesFrom rdf:resource="#ControlConstructList"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="ControlConstructList">
<rdfs:comment> A list of control constructs </rdfs:comment>
  <rdfs:subClassOf rdf:resource="&shadow-rdf;#List"/>
  <rdfs:subClassOf>
   <owl:Restriction>
    <owl:onProperty rdf:resource="&shadow-rdf;#first"/>
    <owl:allValuesFrom rdf:resource="#ControlConstruct"/>
   </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
   <owl:Restriction>
    <owl:onProperty rdf:resource="&shadow-rdf;#rest"/>
    <owl:allValuesFrom rdf:resource="#ControlConstructList"/>
   </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

3.3 Service Grounding

While the service profile and service model, presented in the previous sections, describe what a service provide and its inner design and how it works, the grounding of a service specifies the details of how to access the service. Many different kinds of information are involved: protocol and message formats, serialization, transport, and addressing. The role of grounding is mainly to bridge the gap between semantic description of web services and the existing service description models which are mainly syntactic. In other words, service grounding maps from the more abstract semantic notions to the concrete elements that are necessary for interacting with the service.

The service profile and service model present the abstract side of a service description. However, this abstract description doesn't deal with the messages exchanged during service execution. The only part of a message that is abstractly described is the content, through the description of the input and output properties of the Process class in the Service Model ontology. The service grounding ontology is based on and expands these primitive communication parameters. The main role of a service grounding in OWL-S is to realize process inputs and outputs as messages that are sent and received.

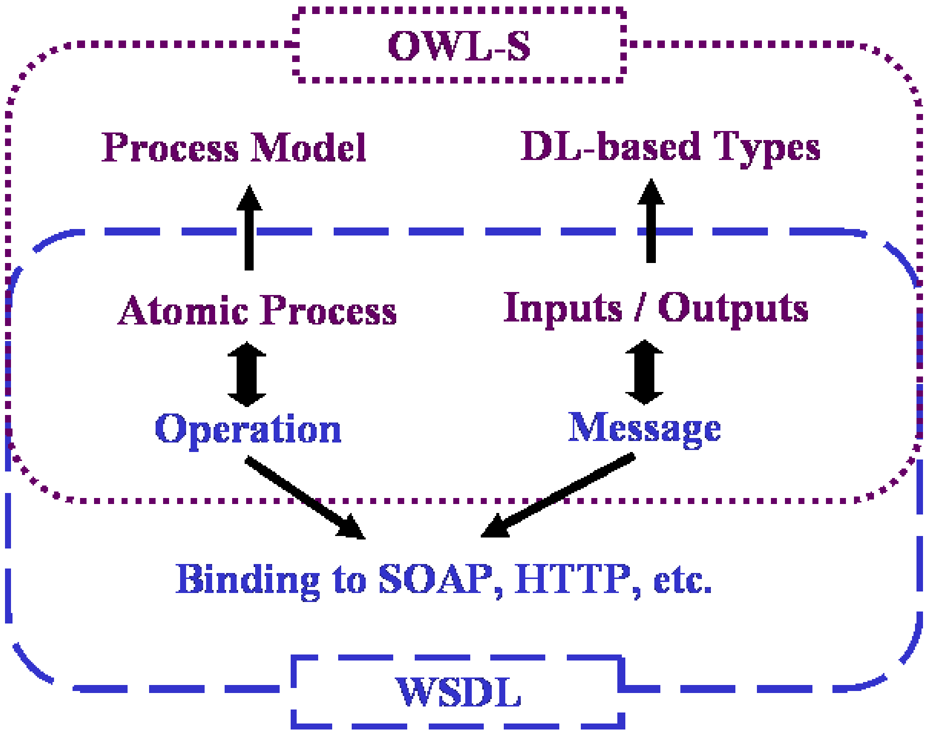
The main objective of the work behind the service grounding is to make use of the existing body of work in the area of concrete message specification which is significant and already standardized in the industry. To that effect, OWL-S makes use of the Web Services Description Language (WSDL) to set an example of an initial grounding mechanism. The purpose here is not to prescribe the only possible grounding approach to be used with all services, but rather to provide a general, canonical and broadly applicable approach that will be useful for the great majority of cases.

A similar concept to that of OWL-S grounding is WSDL's concept of binding. Based on that similarity and the existing WSDL extensibility elements OWL-S grounding using WSDL is realized. For version 1.1 of OWL-S, the version 1.1 WSDL specification is used.

Mapping from WSDL to OWL-S

The work behind service grounding aims to benefit from the advantages of both WSDL and OWL-S. As described in previous sections, the OWL-S process model is an expressive way of describing the inner workings of a service and OWL's typing mechanisms which are based on XML Schema provide the developer with a set of design advantages. On the other hand, the existing description mechanism of WSDL and message declaration and software support of SOAP have been standardized and used extensively, thus they constitute the best available option for declaration of message exchanges. In this subsection, a mapping between OWL-S and WSDL is presented.

The example of a grounding mechanism that involves OWL-S and WSDL that is presented in the OWL-S specification, explicitly states that both languages are required for the full specification of that mechanism. OWL-S alone or WSDL by itself cannot form a complete grounding specification. In the figure shown below, the overlap between the two languages is illustrated. While WSDL defines abstract types specified using XML Schema in order to characterize the inputs and outputs of services, OWL-S allows for the definition of abstract types as OWL classes, based on description logic. Both languages however lack something. On the one hand, WSDL is unable to express the semantics of an OWL class as it is not a semantic language and lacks many required features. On the other hand, OWL-S has no means, as currently defined, to express the binding information that WSDL captures. As a result, both languages are indispensable in a grounding declaration and this enforces the notion of an OWL-S/WSDL grounding that uses OWL classes as the abstract types of message parts declared in WSDL, and then relies on WSDL binding constructs to specify the formatting of the messages.



An OWL-S/WSDL grounding is based upon a series of correspondences between the two languages, as illustrated in the figure above. The rest of this section deals with these correspondences.

An OWL-S atomic process generally corresponds to a WSDL operation. There are four different types of WSDL operations and each one of them can be linked with an atomic process type. Different types of operations are related to OWL-S processes according to the following:

- An atomic process with both inputs and outputs corresponds to a WSDL request-response operation.
- An atomic process with inputs, but no outputs, corresponds to a WSDL one-way operation.
- An atomic process with outputs, but no inputs, corresponds to a WSDL notification operation.

- A composite process with both outputs and inputs, and with the sending of outputs specified as coming before the reception of inputs, corresponds to a WSDL solicit-response operation.

While the most common case is for an atomic process to correspond to a single WSDL operation, there are exceptions. WSDL supports providing multiple definitions with different port types for the same operation. To support this, OWL-S allows for a one-to-many correspondence between an atomic process and multiple WSDL operations. It is also possible, in these situations, to maintain a one-to-one correspondence, by using multiple (differently named) atomic processes.

A second correspondence between OWL-S and WSDL involves inputs and outputs. The set of inputs and the set of outputs of an OWL-S atomic process each correspond to WSDL's concept of message. More precisely, OWL-S inputs correspond to the parts of an input message of a WSDL operation, and OWL-S outputs correspond to the parts of an output message of a WSDL operation.

Finally, a third correspondence between OWL-S and WSDL involves typing. The types of the inputs and outputs of an OWL-S atomic process, which are OWL classes, correspond to the notion of abstract type in WSDL. This means that the WSDL message declaration uses abstract types for typing. These three correspondences between OWL-S and WSDL are what need to be defined when creating a service grounding. It should be noted however that correspondences do not need to be direct. Transformation languages such as XSLT can be used as mediator between WSDL message parts and OWL-S parameters.

The Grounding class

Having described how the relation between OWL-S and WSDL is explored to define a service grounding, our attention is turned to the mechanism by which the relevant WSDL constructs may be referenced in OWL-S. The Grounding class which is a subclass of the ServiceGrounding class deals with that part of the ontology.

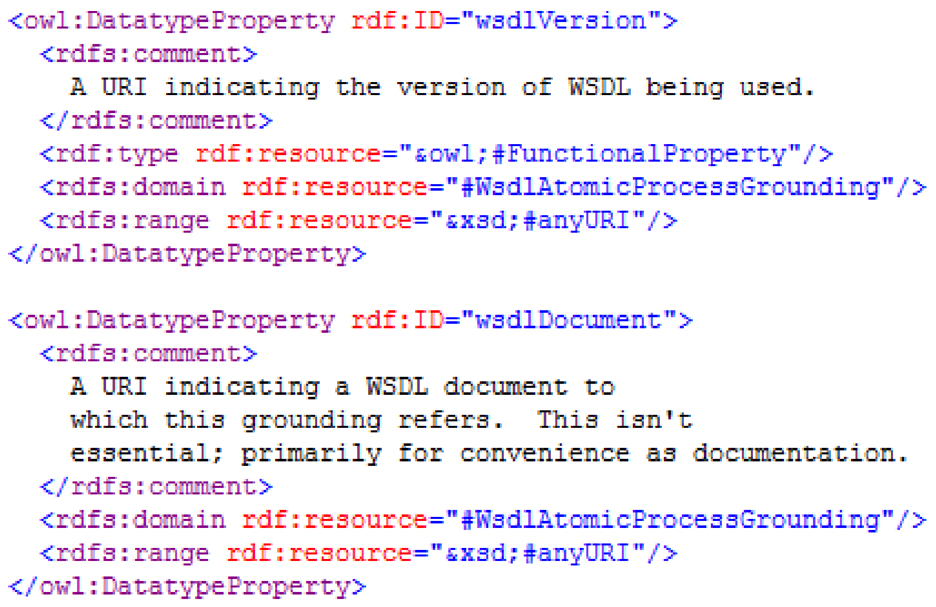
The design of OWL-S allows for multiple types of grounding to be declared. Each of these grounding declarations must be a subclass of the Grounding class (and a subclass of the

ServiceGrounding class due to transitivity). For the WSDL grounding, the OWL-S WsdlGrounding class is declared. Each WsdlGrounding instance should contain a list of instances of the WsdlAtomicProcessGrounding class. The OWL-S code in Grounding.owl is shown below:

```
<owl:Class rdf:ID="Grounding">
  <rdfs:comment>
    A Grounding is just a collection of AtomicProcessGrounding
    instances, one for each atomic process in the process model.
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="&service;#ServiceGrounding"/>
</owl:Class>

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

<owl:Class rdf:ID="WsdlGrounding">
  <rdfs:comment>
    WsdlGrounding is A Grounding that grounds every process to
    a WSDL operation.
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="#Grounding"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasAtomicProcessGrounding"/>
      <owl:allValuesFrom rdf:resource="#WsdlAtomicProcessGrounding"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

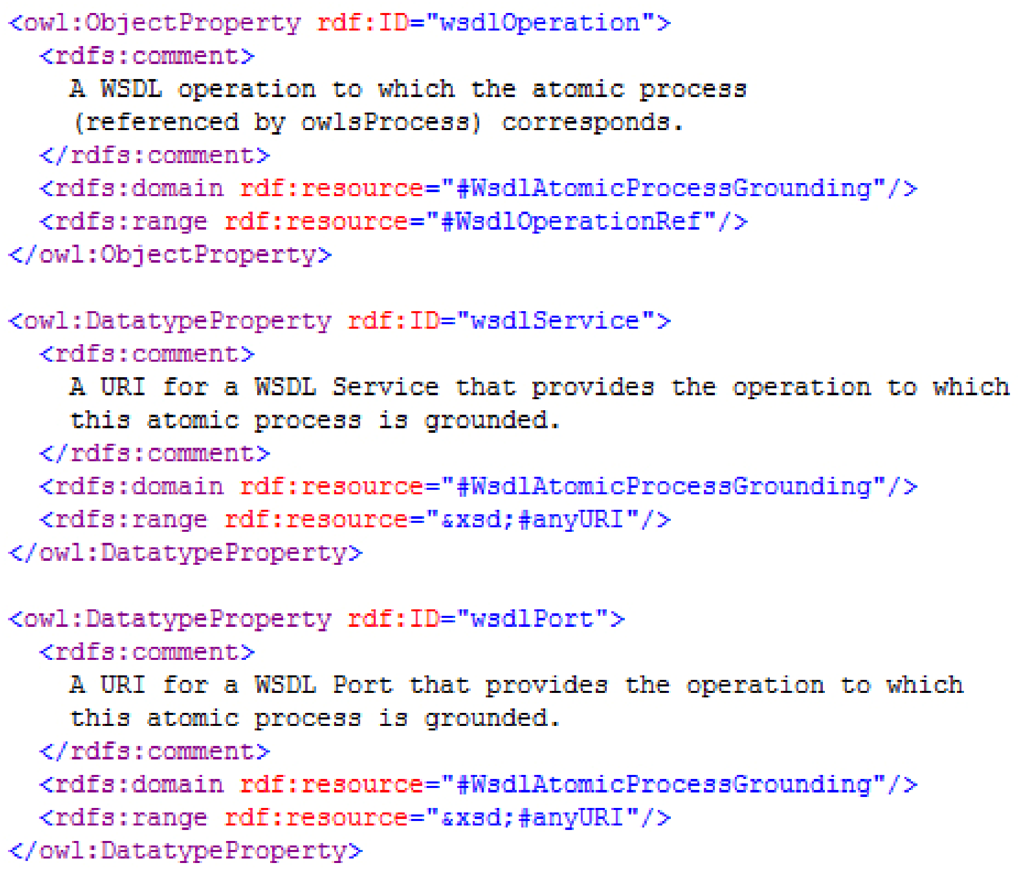
```
<owl:Class rdf:ID="WsdlAtomicProcessGrounding">
  <rdfs:comment>
    A class that relates elements of a OWL-S atomic process to a
    WSDL specification.
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="#AtomicProcessGrounding"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#wsdlOperation"/>
      <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

A WsdlAtomicProcessGrounding instance refers to specific elements within the WSDL specification using a list of properties. wsdlVersion refers to the version of WSDL in use ans is a URI. wsdlDocument points to a URI of the WSDL document to which this grounding instance refers.



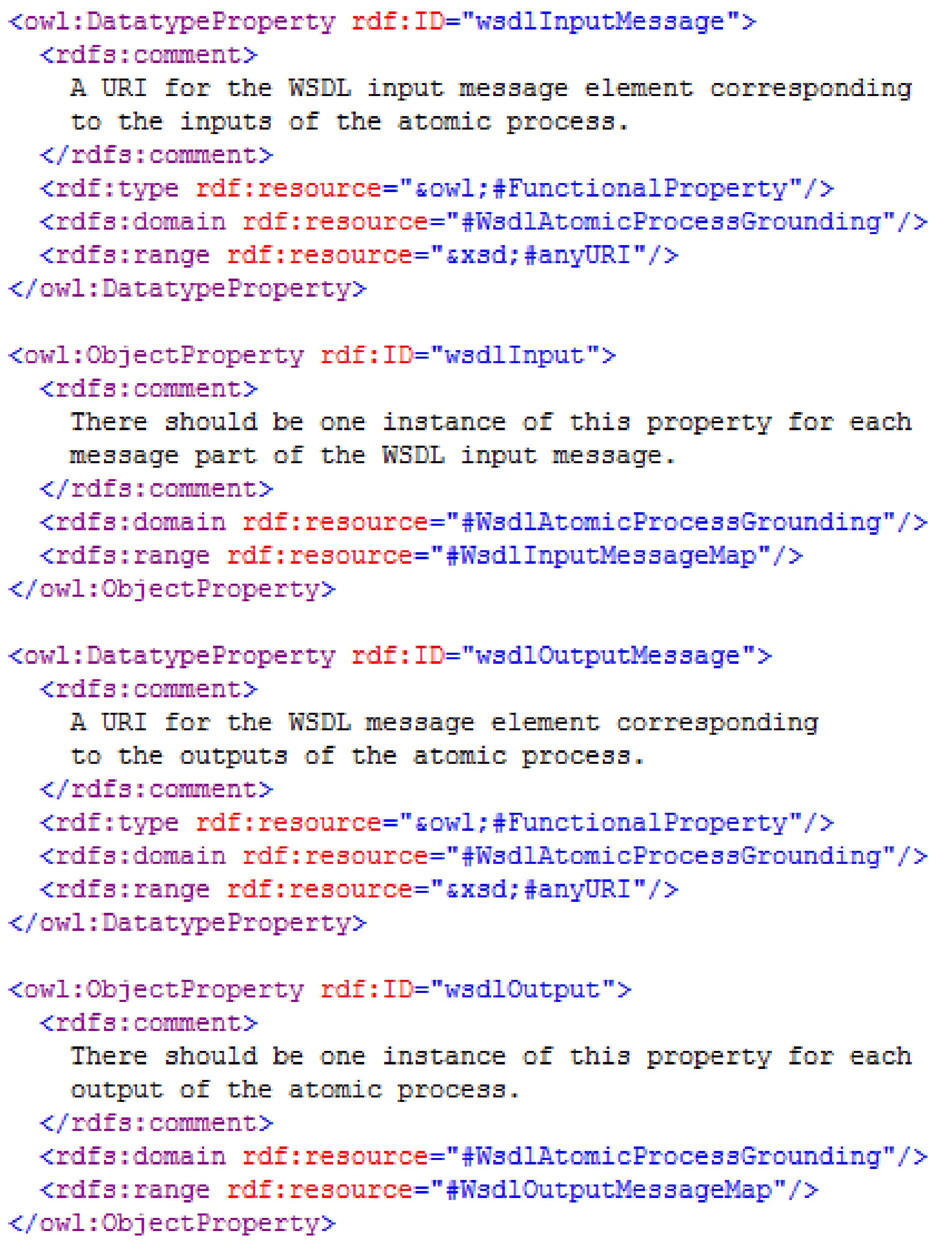
```
<owl:DatatypeProperty rdf:ID="wsdlVersion">
  <rdfs:comment>
    A URI indicating the version of WSDL being used.
  </rdfs:comment>
  <rdf:type rdf:resource="&owl;#FunctionalProperty"/>
  <rdfs:domain rdf:resource="#WsdlAtomicProcessGrounding"/>
  <rdfs:range rdf:resource="&xsd;#anyURI"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="wsdlDocument">
  <rdfs:comment>
    A URI indicating a WSDL document to
    which this grounding refers.  This isn't
    essential; primarily for convenience as documentation.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#WsdlAtomicProcessGrounding"/>
  <rdfs:range rdf:resource="&xsd;#anyURI"/>
</owl:DatatypeProperty>
```

Apart from these two properties, the rest deals directly with the core of a WSDL document which involves operations, ports and messages. wsdlOperation is the URI of the WSDL operation corresponding to the given atomic process. wsdlService is an optional property containing the URI of the WSDL Service that offers the given operation. If we are aware of the port that offers the service and not the service itself, the equivalent wsdlPort property can be used. Both wsdlService and wsdlPort are optional since a wsdlOperation property sometimes is enough to uniquely identify a specified operation. If, however, multiple ports and/or multiple services are offering the specified operation, then the wsdlPort and wsdlService properties are used respectively to uniquely identify the operation. This set of properties is shown in the following code snippet:



```
<owl:ObjectProperty rdf:ID="wsdlOperation">
  <rdfs:comment>
    A WSDL operation to which the atomic process
    (referenced by owlsProcess) corresponds.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#WsdlAtomicProcessGrounding"/>
  <rdfs:range rdf:resource="#WsdlOperationRef"/>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:ID="wsdlService">
  <rdfs:comment>
    A URI for a WSDL Service that provides the operation to which
    this atomic process is grounded.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#WsdlAtomicProcessGrounding"/>
  <rdfs:range rdf:resource="&xsd;#anyURI"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="wsdlPort">
  <rdfs:comment>
    A URI for a WSDL Port that provides the operation to which
    this atomic process is grounded.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#WsdlAtomicProcessGrounding"/>
  <rdfs:range rdf:resource="&xsd;#anyURI"/>
</owl:DatatypeProperty>
```

The rest of the properties of the WsdlAtomicProcessGrounding class deal with messages. wsdlInputMessage contains the URI of the WSDL message definition that carries the inputs of the given atomic process. wsdlOutputMessage contains the URI of the WSDL message definition that carries the outputs of the given atomic process. For each message part of a WSDL message a wsdlInput property is created, containing mapping pairs, instances of the WsdlInputMessageMap class. This mapping is between WSDL message part URIs (expressed with the wsdlMessagePart property) and elements that show how to derive that message part from one or more inputs of the OWL-S atomic process. In the simplest cases -- in which the message part corresponds directly to a single OWL-S input, and the type of the input is directly used by the WSDL specification -- this is done just by mentioning the URI of a particular input object (using the owlsParameter property). In all other cases, the xsltTransformation property gives an XSLT script that generates the message part from an instance of the atomic process. (The script may be given as a string embedded within the grounding instance, or as a URI.) For outputs, the equivalent wsdlOutput property is used. The mappings here are instances of the WsdlOutputMessageMap class. The final set of properties for the WsdlGrounding is shown in the following figures.



```
<owl:DatatypeProperty rdf:ID="wsdlInputMessage">
  <rdfs:comment>
    A URI for the WSDL input message element corresponding
    to the inputs of the atomic process.
  </rdfs:comment>
  <rdf:type rdf:resource="&owl;#FunctionalProperty"/>
  <rdfs:domain rdf:resource="#WsdlAtomicProcessGrounding"/>
  <rdfs:range rdf:resource="&xsd;#anyURI"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="wsdlInput">
  <rdfs:comment>
    There should be one instance of this property for each
    message part of the WSDL input message.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#WsdlAtomicProcessGrounding"/>
  <rdfs:range rdf:resource="#WsdlInputMessageMap"/>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:ID="wsdlOutputMessage">
  <rdfs:comment>
    A URI for the WSDL message element corresponding
    to the outputs of the atomic process.
  </rdfs:comment>
  <rdf:type rdf:resource="&owl;#FunctionalProperty"/>
  <rdfs:domain rdf:resource="#WsdlAtomicProcessGrounding"/>
  <rdfs:range rdf:resource="&xsd;#anyURI"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="wsdlOutput">
  <rdfs:comment>
    There should be one instance of this property for each
    output of the atomic process.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#WsdlAtomicProcessGrounding"/>
  <rdfs:range rdf:resource="#WsdlOutputMessageMap"/>
</owl:ObjectProperty>
```

```
<owl:Class rdf:ID="WsdlMessageMap">
  <rdfs:comment>
    MessageMap for WSDL inputs and outputs
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="#MessageMap"/>
</owl:Class>

<!-- wsdlMessagePart -->

<owl:DatatypeProperty rdf:ID="wsdlMessagePart">
  <rdfs:comment>
    A URI for a WSDL message part element.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#WsdlMessageMap"/>
  <rdfs:range rdf:resource="&xsd;#anyURI"/>
</owl:DatatypeProperty>

<owl:Class rdf:ID="WsdlInputMessageMap">
  <rdfs:subClassOf rdf:resource="#WsdlMessageMap"/>
  <rdfs:subClassOf rdf:resource="#InputMessageMap"/>
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#DirectInputMessageMap"/>
    <owl:Class rdf:about="#XSLTInputMessageMap"/>
  </owl:unionOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#wsdlMessagePart"/>
      <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="WsdlOutputMessageMap">
  <rdfs:subClassOf rdf:resource="#WsdlMessageMap"/>
  <rdfs:subClassOf rdf:resource="#OutputMessageMap"/>
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#DirectOutputMessageMap"/>
    <owl:Class rdf:about="#XSLTOutputMessageMap"/>
  </owl:unionOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#owlsParameter"/>
      <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

4. Why OWL instead of RDFS

The authors chose to use OWL to formally define the ontologies. The choice of a language like OWL is directly linked to the goals the authors set while designing OWL-S. OWL-S is meant to support automatic Web service discovery, invocation, composition and interoperation. The common consensus behind these goals is that the OWL-S ontologies must be machine-interpretable so that agents can reason based on them. OWL provides ontology structuring mechanisms provide an appropriate, Web-compatible representation language framework within which to do this. OWL offers a series of features that are essential for ontologies such as OWL-S and are not supported by other languages such as RDFS.

One important feature that RDFS does not support is expressing cardinality constraints. RDFS cannot express limitations between relationships of subjects and predicates. However, as it should be obvious by now, the service description framework proposed by OWL-S uses cardinality constraints in many properties. For example, in a service profile many properties need to be unique, thus allowing only one property instance per profile instance. This is achieved using the owl:Restriction, owl:onProperty and owl:cardinality elements.

Moreover, OWL allows class expressions such as unionOf, disjointUnionOf, intersectionOf and complementOf. For example, the class declaration of processVar in Process.owl of the OWL-S Service Model uses owl:unionOf to state that it is a collection of instances of the classes Parameter, Existential, Participant, ResultVar and Local. This allows for declarations based on existing declarations, thus facilitating a modular design scheme for the ontologies. In relation to that feature of OWL is the ability to define classes based on property values or other restrictions of an existing class.

Another very useful feature of OWL is inference support. In OWL-S, the authors need to express construct such as inverseOf and disjointWith to model properties that are inverse to each other and disjoint, respectively. For example, the presentedBy property that links a service profile instance to a service instance is the inverse of the presents property that links a service instance to a service profile instance and this is expressed with the owl:inverseOf element.

As a result, OWL supports features that are essential to the OWL-S ontology design and other ones that could be used in further extensions of OWL-S, such as enumerations, equivalence, local restrictions etc.

5. Summary

OWL-S is an exceptional example of how the OWL-based framework of the Semantic Web and the unique features it includes can be used to create rich and complete ontologies, in this case for a semantic description of Web services. Augmenting existing description models for web services with semantic annotations is essential to the Semantic Web and should contribute in the realization of automatic discovery, invocation, composition and monitoring of Web services.

The current version for OWL-S is 1.1, released in November 2004. A prerelease version 1.2 is available as of March 2006 but has not been released fully yet. OWL-S is an ongoing work and should be enhanced in further releases to provide a complete framework for semantic description for Web services.

.

6. Bibliography

1. Ethan Cerami, Web Services Essentials, *O'Reilly & Associates*, February, 2002, Pages 119-154
2. Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1, 2001, Available from http://www.w3.org/TR/2001/NOTE-wsdl-20010315

3. Thomi Pilioura, Aphrodite Tsalgatidou, Alexandros Batsakis, Using WSDL/UDDI and DAML-S in Web Service Discovery, Proceedings of WWW 2003 Workshop on E-Services and the Semantic Web (ESSW' 03), Budapest, Hungary, 2003
4. Liliana Cabral, John Domingue, Enrico Motta, Terry Payne and Farshad Hakimpour, Approaches to Semantic Web Services: An Overview and Comparisons, In proceedings of the First European Semantic Web Symposium (ESWS2004), Heraklion, Crete, Greece, May, 2004.
5. Naveen Balani, The future of the Web is Semantic, Available from http://www.ibm.com/developerworks/web/library/wa-semweb/#2
6. Stefan Tang, Matching Of Web Service Specifications Using Daml-S Descriptions, PHD Thesis, Technische Universität Berlin, March, 2004
7. Grigoris Antoniou, Enrico Franconi, and Frank van Harmelen, Introduction to Semantic Web Ontology Languages, Reasoning Web, LNCS 3564, Springer 2005: 1-21, 2005
8. David Martin, Massimo Paolucci, Sheila McIlraith, Mark Burstein, Drew McDermott, Deborah McGuinness, Bijan Parsia, Terry Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia Sycara, OWL-S: Semantic Markup for Web Service, Available from http://www.w3.org/Submission/OWL-S/
9. Natenapa Sriharee and Twittie Senivongse, Discovering Web Services Using Behavioural Constraints and Ontology, Proceedings of Distributed Applications and Interoperable Systems, 2003
10. Anupriya Ankolekar, David Martin, Deborah McGuinness, Sheila McIlraith, Massimo Paolucci, Bijan Parsia, OWL-S' Relationship to Selected Other Technologies, Available from http://www.daml.org/services/owl-s/1.1/related.html
11. Michael Stollberg, Cristina Feier, Dumitru Roman, and Dieter Fensel Semantic Web Services - Concepts and Technologies, 7th Edition of the EUROLAN Summer School "The Multilingual Web: Resources, Technologies, and Prospects", Cluj-Napoca, Romania, August 2, 2005.