OWL-S: Semantic Markup for Web Services

The OWL Services Coalition*

Abstract

The Semantic Web should enable greater access not only to content but also to services on the Web. Users and software agents should be able to discover, invoke, compose, and monitor Web resources offering particular services and having particular properties. As part of the DARPA Agent Markup Language program, we are developing an ontology of services, called OWL-S (formerly DAML-S), that makes these functionalities possible. In this white paper we describe the overall structure of the ontology and its three main parts: the service *profile* for advertising and discovering services; the *process model*, which gives a detailed description of a service's operation; and the *grounding*, which provides details on how to interoperate with a service, via messages. We also discuss the motivation for OWL-S, and work on related ontologies for resources and for time.

This white paper accompanies OWL-S version 1.1, which is available at http://www.daml.org/services/.

1 Introduction: Services on the Semantic Web

Efforts toward the creation of the Semantic Web are gaining momentum [1]. Soon it will be possible to access Web resources by content rather than just by keywords. A significant force in this movement is the development of a new generation of Web markup languages such as OWL[16] and its predecessor DAML+OIL [7, 9]. These languages enable the creation of ontologies for any domain and the instantiation of these ontologies in the description of specific Web sites.

Among the most important Web resources are those that provide services. By "service" we mean Web sites that do not merely provide static information but allow one to effect some action or change in the world, such as the sale of a product or the control of a physical device. The Semantic Web should enable users to locate, select, employ, compose, and monitor Web-based services automatically.

To make use of a Web service, a software agent needs a computer-interpretable description of the service, and the means by which it is accessed. An important goal for Semantic Web markup languages, then, is to establish a framework within which these descriptions are made and shared. Web sites should be able to employ a set of basic classes and properties for declaring

*This work is the collaborative effort of projects and individuals at BBN Technologies, Carnegie-Mellon University, De Montfort University, Nokia, Stanford University, SRI International, University of Maryland at College Park, University of Southampton, University of Toronto, USC Information Sciences Institute, Vrije Universiteit Amsterdam, and Yale University. Mark Burstein participates at BBN Technologies. The participants at Carnegie-Mellon University are Anupriya Ankolenkar, Massimo Paolucci, Naveen Srinivasan, and Katia Sycara. Monika Solanki is at De Montfort University, and Ora Lassila participates for Nokia. Deborah McGuinness is at Stanford University, and the participants for SRI International are Grit Denker and David Martin. Bijan Parsia and Evren Sirin are at the University of Maryland at College Park, Terry Payne at the University of Southampton, Sheila McIlraith at the University of Toronto, Jerry Hobbs at USC ISI, Marta Sabou at Vrije Universiteit Amsterdam, and Drew McDermott at Yale.

and describing services, and the ontology structuring mechanisms of OWL provide the appropriate framework within which to do this.

This paper describes a collaborative effort by researchers at several organizations to define just such an ontology. We call this language OWL-S[1]. We first motivate our effort with some sample tasks. In the central part of the paper we describe the upper ontology for services that we have developed, including its subontologies for profiles, processes, and groundings. The ontology is still evolving, and making connections to other development efforts, such as those building ontologies of time and resources.

This paper accompanies OWL-S version 1.1, which is available at [3]. Please note that, in addition to the OWL ontology files, the release site includes examples and additional forms of documentation, including, in particular, a code walk-through illustrative of many points in this document, additional explanatory material (in HTML) regarding the grounding and the use of profile-based class hierarchies, and information about the status of this work, including unresolved issues and future directions.

2 Some Motivating Tasks

Services can be simple, or "primitive," in the sense that they invoke only a single Web-accessible computer program, sensor, or device that does not rely upon another Web service, and there is no ongoing interaction between the user and the service, beyond a simple response. For example, a service that returns a postal code or the longitude and latitude when given an address would be in this category. Alternately, services can be complex, composed of multiple primitive services, often requiring an interaction or conversation between the user and the services, so that the user can make choices and provide information conditionally. One's interaction with www.amazon.com to buy a book is like this; the user searches for books by various criteria, perhaps reads reviews, may or may not decide to buy, and gives credit card and mailing information. OWL-S is meant to support both categories of services, but complex services have provided the primary motivations for the features of the language. The following four task types will give the reader an idea of the kinds of tasks we expect OWL-S to enable [17, 18].

1. **Automatic Web service discovery.** Automatic Web service discovery involves the automatic location of Web services that provide a particular service and that adhere to requested constraints. For example, the user may want to find a service that sells airline tickets between two given cities and accepts a particular credit card. Currently, this task must be performed by a human who might use a search engine to find a service, read the Web page, and execute the service manually, to determine if it satisfies the constraints. With OWL-S markup of services, the information necessary for Web service discovery could be specified as computer-interpretable semantic markup at the service Web sites, and a service registry or ontology-enhanced search engine could be used to locate the services automatically. Alternatively, a server could proactively advertise itself in OWL-S with a service registry, also called middle agent [4, 25, 15], so that requesters can find it when they query the registry. Thus, OWL-S must provide declarative advertisements of service properties and capabilities that can be used for automatic service discovery.

2. **Automatic Web service invocation.** Automatic Web service invocation involves the automatic execution of an identified Web service by a computer program or agent. For example, the user could request the purchase, from a particular site, of an airline ticket

[1] Originally called DAML-S.

on a particular flight. Currently, a user must go to the Web site offering that service, fill out a form, and click on a button to execute the service. Alternately, the user might send an HTTP request directly to the service with the appropriate parameters in HTML. In either case, a human in the loop is necessary. Execution of a Web service can be thought of as a collection of function calls. OWL-S markup of Web services provides a declarative, computer-interpretable API for executing these function calls. A software agent should be able to interpret the markup to understand what input is necessary to the service call, what information will be returned, and how to execute the service automatically. Thus, OWL-S must provide declarative APIs for Web services that are necessary for automated Web service execution.

3. **Automatic Web service composition and interoperation.** This task involves the automatic selection, composition, and interoperation of Web services to perform some task, given a high-level description of an objective. For example, the user may want to make all the travel arrangements for a trip to a conference. Currently, the user must select the Web services, specify the composition manually, and make sure that any software needed for the interoperation is custom-created. With OWL-S markup of Web services, the information necessary to select and compose services will be encoded at the service Web sites. Software can be written to manipulate these representations, together with a specification of the objectives of the task, to achieve the task automatically. Thus, OWL-S must provide declarative specifications of the prerequisites and consequences of individual service use that are necessary for automatic service composition and interoperation.

4. **Automatic Web service execution monitoring.** Individual services and, even more, compositions of services, will often require some time to execute completely. A user may want to know during this period what the status of his or her or request is, or plans may have changed, thus requiring alterations in the actions the software agent takes. For example, a user may want to make sure that a hotel reservation has already been made. For these purposes, it would be good to have the ability to find out where in the process the request is and whether any unanticipated glitches have appeared. Thus, OWL-S should provide declarative descriptors for the state of execution of services.

We include the last task type because we think it is important, but in fact versions of OWL-S developed so far have not ventured into this area. Our main focus has been on the other three task types.

Any Web-accessible program/sensor/device that is *declared* as a service will be regarded as a service. OWL-S does not preclude declaring simple, static Web pages to be services. But our primary motivation in defining OWL-S has been to support more complex tasks of the kinds described above.

3 An Upper Ontology for Services

Our structuring of the ontology of services is motivated by the need to provide three essential types of knowledge about a service (shown in figure 1), each characterized by the question it answers:

- *What does the service require of the user(s), or other agents, and provide for them?* The answer to this question is given in the "profile[2]." Thus, the class SERVICE *presents* a

[2]Service profile has also been called "service capability advertisement" [23].

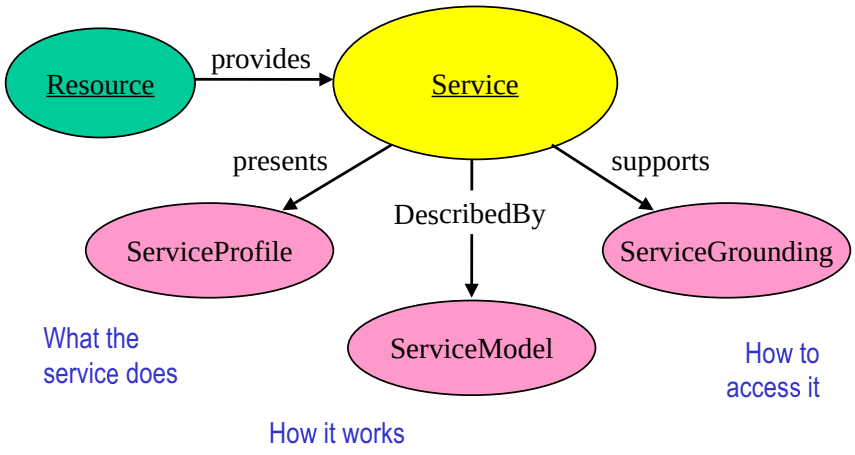

Figure 1: Top level of the service ontology

SERVICEPROFILE

- *How does it work?* The answer to this question is given in the "model." Thus, the class SERVICE is *describedBy* a SERVICEMODEL
- *How is it used?* The answer to this question is given in the "grounding." Thus, the class SERVICE *supports* a SERVICEGROUNDING.

The class SERVICE provides an organizational point of reference for declaring Web services; one instance of SERVICE will exist for each distinct published service. The properties *presents*, *describedBy*, and *supports* are properties of SERVICE. The classes SERVICEPROFILE, SERVICEMODEL, and SERVICEGROUNDING are the respective ranges of those properties. Each instance of SERVICE will *present* a descendant class of SERVICEPROFILE, be *describedBy* a descendant class of SERVICEMODEL, and *support* a descendant class of SERVICEGROUNDING. The details of profiles, models, and groundings may vary widely from one type of service to another—that is, from one descendant class of SERVICE to another. But each of these three classes provides an essential type of information about the service, as characterized in the rest of the paper.

The service profile tells "what the service does"; that is, it gives the types of information needed by a service-seeking agent (or matchmaking agent acting on behalf of a service-seeking agent) to determine whether the service meets its needs. In addition to representing the capabilities of a service, the profile can be used to express the needs of the service-seeking agent, so that a matchmaker has a convenient dual-purpose representation upon which to base its operations.

The service model tells "how the service works"; that is, it describes what happens when the service is carried out. For nontrivial services (those composed of several steps over time), this

description may be used by a service-seeking agent in at least four different ways: (1) to perform a more in-depth analysis of whether the service meets its needs; (2) to compose service descriptions from multiple services to perform a specific task; (3) during the course of the service enactment, to coordinate the activities of the different participants; and (4) to monitor the execution of the service.

A service grounding ("grounding" for short) specifies the details of how an agent can access a service. Typically a grounding will specify a communication protocol, message formats, and other service-specific details such as port numbers used in contacting the service. In addition, the grounding must specify, for each abstract type specified in the ServiceModel, an unambiguous way of exchanging data elements of that type with the service (that is, the serialization techniques employed).

Generally speaking, the ServiceProfile provides the information needed for an agent to discover a service. Taken together, the ServiceModel and ServiceGrounding objects associated with a service provide enough information for an agent to make use of a service.

The upper ontology for services specifies only two cardinality constraints: a service can be described by at most one service model, and a grounding must be associated with exactly one service. The upper ontology deliberately does not specify any minimum cardinality for the properties *presents* or *describedBy*. (Although, in principle, a service needs all three properties to be fully characterized, it is easy to imagine situations in which a partial characterization could be useful.) Nor does the upper ontology specify any maximum cardinality for *presents* or *supports*. (It will be extremely useful for some services to offer multiple profiles and/or multiple groundings.)

Finally, it must be noted that while we define one particular upper ontology for profiles, one for service models, and one for groundings, nevertheless OWL-S allows for the construction of alternative approaches in each case. Our intent here is *not* to prescribe a single approach in each of the three areas, but rather to provide default approaches that will be useful for the majority of cases. In the following three sections we discuss the resulting service profile, service model, and service grounding in greater detail.

4 Service Profiles

A transaction in a web services marketplace involves three parties: the service requesters, the service provider, and infrastructure components [24, 25]. The service requester, which may broadly identify with the buyer, seeks a service to complete its work; the service provider, which can be broadly identified with the seller, provides a service sought by the requester. In an open environment such as the Internet, the requester may not know ahead of time of the existence of the provider, so the requester relies on infrastructure components that act like registries to find the appropriate provider. For instance, a requester may need a news service that reports stock quotes with no delay with respect to the market. The role of the registries is to match the request with the offers of service providers to identify which of them is the best match. Within the OWL-S framework, the Service Profile provides a way to describe the services offered by the providers, and the services needed by the requesters.

The Service Profile does not mandate any representation of services; rather, using the OWL subclassing it is possible to create specialized representations of services that can be used as service profiles. OWL-S provides one possible representation through the class Profile. An OWL-S Profile describes a service as a function of three basic types of information: what organization provides the service, what function the service computes, and a host of features that specify characteristics of the service. The three pieces of information are reviewed in order below.

The provider information consists of contact information that refers to the entity that provides the service. For instance, contact information may refer to the maintenance operator that is responsible for running the service, or to a customer representative that may provide additional information about the service.

The functional description of the service is expressed in terms of the transformation produced by the service. Specifically, it specifices the inputs required by the service and the outputs generated; furthermore, since a service may require external conditions to be satisfied, and it has the effect of changing such conditions, the profile describes the preconditions required by the service and the expected effects that result from the execution of the service. For example, a selling service may require as a precondition a valid credit card and as input the credit card number and expiration date. As output it generates a receipt, and as effect the card is charged.

Finally, the profile allows the description of a host of properties that are used to describe features of the service. The first type of information specifies the category of a given service, for example, the category of the service within the UNSPSC classification system. The second type of information is quality rating of the service: some services may be very good, reliable, and quick to respond; others may be unreliable, sluggish, or even malevolent. Before using a service, a requester may want to check what kind of service it is dealing with; therefore, a service may want to publish its rating within a specified rating system, to showcase the quality of service it provides. It is up to the service requester to use this information, to verify that it is indeed correct, and to decide what to do with it. The last type of information is an unbounded list of service parameters that can contain any type of information. The OWL-S Profile provides a mechanism for representing such parameters; which might include parameters that provide an estimate of the max response time, to the geographic availability of a service.

4.1 Compiling a Profile: The Relation with Process Model

The Profile of a service provides a concise description of the service to a registry, but once the service has been selected the Profile is useless; rather, the client will use the Process Model to control the interaction with the service. Although the Profile and the Process Model play different roles during the transaction between Web services, they are two different representations of the same service, so it is natural to expect that the input, output, precondition, and effects (hereafter IOPEs) of one are reflected in the IOPEs of the other.

OWL-S does not dictate any constraint between Profiles and Process Models, so the two descriptions 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. As an extreme example, imagine a service that advertises as a travel agent, but adopts the process model of a book selling agent; it will be selected to reserve travels, but it will fail to do that, asking instead for book titles and ISBN numbers. On the other side, it will never be selected by services that want to buy books, so it will never sell a book either.

The selection of the IOPEs to specify in the Profile is quite a tricky process. It should avoid misrepresentation of the service, so ideally it would require all the IOPEs used in the Process Model. On the other side, some of those IOPEs may be so general that they do not describe the service. Another thing to consider is the registry's algorithm for matching requests with providers. Furthermore, the Profile implicitly specifies the intended purpose of the service: it advertises those functionalities that the service wants to provide, while it may hide (not declare publicly) other functionalities. As an example, consider a book-selling service that may involve two functionalities: the first one allows other services to browse its site to find books of interest,

and the second one allows users to buy the books they found. The book seller has the choice of advertising just the book-buying functionality or both the browsing functionality and the buying functionality. In the latter case, the service makes public the fact that it can provide browsing services, and it allows everybody to browse its registry without buying a book. In contrast, by advertising only the book-selling functionality, but not the browsing, the agent discourages browsing by requesters who do not intend to buy. The decision as to which functionalities to advertise determines how the service will be used: a requester who intends to browse but not to buy would select a service that advertises both buying and browsing capabilities, but not one that advertises buying only.

In the description so far, we tacitly assumed a registry model in which service capabilities are advertised, and then matched against requests of service. This is the model adopted by registries like UDDI. While this is the most likely model to be adopted by Web services, other forms of registry are also possible. For example, when the demand for a service is higher than the supply, then advertising needs for service is more efficient then advertising offered services since a provider can select the next request as soon as it is free; furthermore, in a pure P2P architecture there would be no registry at all. Indeed the types of registry may vary widely and as many as 28 different types have been identified [25, 4]. By using a declarative representation of Web services, the service profile is not committed to any form of registry, but it can be used in all of them. Since the service profile represents both offers of services and needs of services, then it can be used in a reverse registry that records needs and queries on offers. Indeed, the Service Profile can be used in all 28 types of registry.

4.2 Profile Properties

In the following we describe in detail the main parts of the profile model; we classify them into four sections: the first one (4.2.1) describes the properties that link the Service Profile class with the Service class and Process Model class; the second section (4.2.2) describes the form of contact information and the Description of the profile — this is information usually intended for human consumption; in the third section (4.2.3), we discuss the functional representation in terms of IOPEs; finally, in the last section (4.2.4), we describe the attributes of the Profile.

4.2.1 Service Profile

The class ServiceProfile provides a superclass of every type of high-level description of the service. ServiceProfile does not mandate any representation of services, but it mandates the basic information to link any instance of profile with an instance of service.

There is a two-way relation between a service and a profile, so that a service can be related to a profile and a profile to a service. These relations are expressed by the properties `presents` and `presentedBy`.

presents describes a relation between an instance of service and an instance of profile, it basically says that the service is described by the profile.

presentedBy is the inverse of `presents`; it specifies that a given profile describes a service.

4.2.2 Service Name, Contacts and Description

Some properties of the profile provide human-readable information that is unlikely to be automatically processed. These properties include `serviceName`, `textDescription` and `contactInformation`.

A profile may have at most one service name and text description, but as many items of contact information as the provider wants to offer.

serviceName refers to the name of the service that is being offered. It can be used as an identifier of the service.

textDescription provides a brief description of the service. It summarizes what the service offers, it describes what the service requires to work, and it indicates any additional information that the compiler of the profile wants to share with the receivers.

contactInformation provides a mechanism of refering to humans or individuals responsible for the service (or some aspect of the service). The range of this property is unspecified withing OWL-S, but can be restricted to some other ontology, such as FOAF, VCard, or the now depreciated Actor class provided in previous versions of DAML-S.

4.2.3 Functionality Description

An essential component of the profile is the specification of what functionality the service provides and the specification of the conditions that must be satisfied for a successful result. In addition, the profile specifies what conditions result from the service, including the expected and unexpected results of the service activity. The OWL-S Profile represents two aspects of the functionality of the service: the information transformation (represented by inputs and outputs) and the state change produced by the execution of the service (represented by preconditions and effects). For example, to complete the sale, a book-selling service requires as input a credit card number and expiration date, but also the precondition that the credit card actually exists and is not overdrawn. The result of the sale is the output of a receipt that confirms the proper execution of the transaction, and as effect the transfer of ownership and the physical transfer of the book from the the warehouse of the seller to the address of the buyer.

The Profile ontology does not provide a schema to describe IOPE instances. However, such a schema exists in the Process ontology, as discussed in the next section. Ideally, we envision that the IOPE's published by the Profile are a subset of those published by the Process. Therefore, the Process part of a description will create all the IOPE instances and the Profile instance can simply point to these instances. In this case a single instance is created for any IOPE, unlike in previous versions of OWL-S when, for a certain IOPE, an instance was created both in the Profile and Process part of the OWL-S description. However, if the IOPE's of the Profile are different from those of the Process, the Profile can still create its own IOPE instances using the schema offered by the Process ontology.

The Profile ontology defines the following properties of the Profile class for pointing to IOPE's:

hasParameter ranges over a Parameter instance of the Process ontology. Note that the Parameter class models our intuition that Inputs and Outputs (which are kinds of Parameters) are both involved in information transformation and therefore they are different from Preconditions and Effects. As a consequence, we do not expect this class to be instantiated. It's role is solely making domain knowledge explicit.

hasInput ranges over an Input instance.

hasOutput ranges over instances of type ConditionalOutput, as defined in the Process ontology. UnConditionalOutput is a subclass of ConditionalOutput therefor this property can range over instances of outputs which are not conditioned.

hasPrecondition specifies one of the preconditions of the service and ranges over a Precondition instance defined according to the schema in the Process ontology.

hasEffect specifies one of the effects of the service. It takes as value an instance of a ConditionalEffect. Similar to outputs, UnConditionalEffect is a subclass of ConditionalEffect and therefore the hasEffect property allows us to refer to simple effects as well.

4.2.4 Profile Attributes

In the previous section we introduced the functional description of services, but there are other aspects of services of which users should be aware. These additional attributes include the quality guarantees that are provided by the service, possible classification of the service, and additional parameters that the service may want to specify.

serviceParameter is an expandable list of properties that may accompany a profile description. The value of the property is an instance of the class ServiceParameter (4.2.5).

serviceCategory refers to an entry in some ontology or taxonomy of services. The value of the property is an instance of the class ServiceCategory (4.2.6)

4.2.5 ServiceParameter

serviceParameterName is the name of the actual parameter, which could be just a literal, or perhaps the URI of the process parameter (a property).

sParameter points to the value of the parameter within some OWL ontology.

4.2.6 ServiceCategory

ServiceCategory describes categories of services on the bases of some classification that may be outside OWL-S and possibly outside OWL. In the latter case, they may require some specialized reasoner if any inference has to be done with it.

categoryName is the name of the actual category, which could be just a literal, or perhaps the URI of the process parameter (a property).

taxonomy stores a reference to the taxonomy scheme. It can be either a URI of the taxonomy, or a URL where the taxonomy resides, or the name of the taxonomy or anything else.

value points to the value in a specific taxonomy There may be more than one value for each taxonomy, so no restriction is added here.

code to each type of service stores the code associated to a taxonomy.

5 Modeling Services as Processes

To give a detailed perspective on how to interact with a service, it can be viewed as a *process*. Specifically, OWL-S 1.1 defines a subclass of `ServiceModel`, `Process`, which draws upon well-established work in a variety of fields, including work in AI on standardizations of planning languages [6], work in programming languages and distributed systems [20, 19], emerging standards in process modeling and workflow technology such as the NIST's Process Specification Language

(PSL) [22] and the Workflow Management Coalition effort (http://www.aiim.org/wfmc), work on modeling verb semantics and event structure [21], previous work on action-inspired Web service markup [18], work in AI on modeling complex actions [13], and work in agent communication languages [15, 5].

It is important to understand that a process is not a program to be executed. It is a specification of the ways a client may interact with a service. An *atomic* process is a description of a service that expects one (possibly complex) message and returns one (possibly complex) message in response. A *composite* process is one that maintains some state; each message the client sends advances it through the process.

A 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, 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 returns. There can be any number of preconditions, which must all hold in order for the process to be 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. (We use the term *perform* instead of *execute* to de-emphasize the traditional picture of a single agent being responsible for the occurrence of the process.)

Before we can go into the details of how processes work, it's necessary to explain how inputs, outputs, conditions, and effects (colloquially known as IOPEs) work, because fitting them into the OWL framework requires bending the rules somewhat.

5.1 Parameters and Expressions

Inputs and outputs are subclasses of a general class called `Parameter`. It's convenient to identify parameters with what are called *variables* in SWRL, the language for expressing OWL Rules.

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

Every parameter has a type, 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.

```
<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>
```

Inputs and outputs 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>
```

Two other subclasses of `Parameter`, `Local` and `ResultVar`, are described below.

Modeling variables as global, named individuals, as is done for Owl Rules, can be misleading. RDF has no notion of the "scope" of a variable, because an RDF document is nothing but a pile of triples. A variable is named with a URI, like any other resource, and so has global scope, or, more accurately, no notion of scope at all. In spite of this lack of structure, we often use RDF to encode hierarchical entites such as formulas and control structures. Wrapping variable references inside literals allows us to sneak in and impose our own scoping rules. We discuss the OWL-S rules in section 5.2.

A process cannot be invoked unless its *preconditions* are true. If and when it does get triggered, it has various *effects*. For example, an agent can order 1000 bolts from a web service only if it can get the web service to accept its promise to pay. One effect of placing the order is the transfer of ownership of the bolts from the service to the agent (or the legal person for which it is a proxy).

Preconditions and effects are represented as logical formulas. Getting logical formulas into RDF has not been easy, but it is now reasonably clear how to proceed. There are actually several possible approaches, depending on how close to RDF/OWL one wants to remain. Usually having lots of choices for such a crucial job is a bad idea, but in this case most of the differences are superficial; it is fairly easy to translate between alternative notations.

The key idea underpinning our approach is to treat expressions as literals, either string literals or XML literals. The latter case is used for languages whose standard encoding is in XML, such as SWRL [8] or RDF [11]. The former case is for other languages such as KIF [10] and PDDL [6]. The ontology [`http://www.daml.org/services/owl-s/1.1/generic/Expression.owl`] defines `Expression`s and their properties.

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

We annotate expressions with the language they are expressed in:

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

The `expressionBody` property gives the actual expression:

```
<owl:DatatypeProperty rdf:ID="expressionBody">
  <rdfs:domain rdf:resource="#Expression"/>
</owl:DatatypeProperty>
```

As an example, we might state that in order to send the number of a certain credit card to a web agent, one must know what its number is:

```
  <Description rdf:about="#process2">
     <hasPrecondition>
        <Expression expressionLanguage="&expr;#KIF">
           <process:expressionBody>
       (!agnt:know_val_is
    (!ecom:credit_card_num ?cc)
    ?num)
   </process:expressionBody>
        </Expression>
     </hasPrecondition>
  </Description>
```

(where the notation "`!ecom:`" for namespaces is taken from Lassila's WILBUR system [12]). (The declaration of the variable `?cc` is not shown. We'll come back to that point shortly.)

In cases where an XML encoding is used, we would declare an expression to be an XML literal. Here's the same example using DRS as the expression language:

```
  <Description rdf:about="#process2">
     <hasPrecondition>
        <Expression expressionLanguage="&drs;#DRS">
   <process:expressionBody>
      <drs:Atomic_formula>
 <rdf:predicate rdf:resource="&agnt;#Know_val_is"/>
 <rdf:subject>
    <drs:Functional_term>
       <drs:function rdf:resource="&ecom;credit_card_num"/>
       <drs:term_args rdf:parseType="Collection">
  <swrl:Variable rdf:resource="#CC"/>
       </drs:term_args>
    </drs:Functional_term>
 </rdf:subject>
 <rdf:object rdf:resource="#Num"/>
      </drs:Atomic_formula>
   </process:expressionBody>
        </Expression>
     </hasPrecondition>
  </Description>
```

The references to `#CC` and `#Num` in the DRS example are to parameters, the same ones written as `?cc` and `?num` in the KIF example. We haven't yet provided a mechanism for declaring the

scopes of variables (see section 5.2) and how they acquire values. In the example, `#CC` is an input parameter to the process, that is, supplied by the client, but `#Num` is supposed to be set in the process of reasoning about this very precondition. Verifying that the process is feasible requires retrieving the credit-card's 16-digit number, which is then associated with the variable `#Num`. We call such a parameter a *local* parameter.

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

The three types of parameter are disjoint:

```
<rdf:Description rdf:about="#Input">
  <owl:disjointWith rdf:resource="#Output"/>
  <owl:disjointWith rdf:resource="#Local"/>
</rdf:Description>

<rdf:Description rdf:about="#Output">
  <owl:disjointWith rdf:resource="#Local"/>
</rdf:Description>
```

Of course, flagging bits of RDF as "Literals" means that an RDF parser should ignore them. (If it did not, then it might turn the RDF into a set of triples with a simple declarative meaning, which is not appropriate.) The trick is to have the OWL-S parser extract the ignored stuff and interpret it appropriately for its context, treating it as ordinary RDF after transformations such as replacing occurrences of variables with their values. In the example above, the occurrences of `#num` and `#cc` are interpreted as the values of these variables, not the variables themselves. In the KIF example, the expressions `?num` and `?cc` must be similarly interpreted. It is usually not too difficult to do this sort of "field engineering" to interface an assertional language to RDF.

There are two special cases of `Expression`: `Condition` and `Effect`. Because they are implemented as literals, there is no way to declare what this difference is, but it's a useful distinction for a human reader of the ontology.

```
<owl:Class rdf:ID="Condition">
   <owl:subClassOf rdf:resource="\&expr;#Expression"/>
</owl:Class>

<owl:Class rdf:ID="Effect">
   <owl:subClassOf rdf:resource="\&expr;#Expression"/>
</owl:Class>
```

5.2 Process Parameters and Results

We connect processes to their "IOPEs" using the properties shown in this table:

Property	*Range*	*Kind*
`hasParticipant`	Agent	`Parameter`
`hasInput`	`Input`	*Parameter*
`hasOutput`	`Output`	*Parameter*
`hasLocal`	`Local`	*Parameter*
`hasPrecondition`	`Condition`	*Expression*
`hasResult`	`Result`	(see below)

As promised above, the links from a process to its parameters implicitly gives them *scope*. Participant, input, output, and local parameters have as scope the entire process they occur in. We introduce *result vars* below, which have a narrower scope.

In the rest of this section, we will discuss the entries in this table.

- **Participants**

  A process involves two or more agents. One is `TheClient`, the agent from whose point of view the process is described. Another is `theServer`, the principal element of the service that the client deals with. If there are others, they are listed using the property `hasParticipant`.

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

<owl:ObjectProperty rdf:ID="hasClient">
  <rdfs:subPropertyOf rdf:resource="#hasParticipant"/>
</owl:ObjectProperty>

<process:Parameter rdf:ID="TheClient">
<process:Parameter rdf:ID="TheServer">
```

- **Inputs and Outputs**

  Inputs and outputs specify the data transformation produced by the process. Inputs specify the information that the process requires for its execution. For atomic processes, the information must come from the client. For the pieces of a composite process, some inputs come directly from the client, but others come from previous steps of the process.

  We said above that an atomic process corresponds to a one-step service that expects one message, so it might appear to be contradictory to allow an atomic process to have multiple inputs. The contradiction is resolved by distinguishing between the *inputs* and the *message* sent to a process. There is just one message, but it can bundle as many inputs as required. The bundling is specified by the *grounding* of the process model; see section 6. Similarly, the outputs produced by the invocation of an atomic process flow back to the client as a single message, the format of which is specified by the grounding. (Here we refer to the WSDL-based grounding, which is the only style of grounding developed to date.)

  The following example shows the definition of `hasParameter`, and its subproperties `hasInput`, `hasOutput`, and `hasLocal`:

```
<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>

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

- **Preconditions and Results**

  If a process has a precondition, it cannot be performed successfully unless the precondition is true.

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

  Please be sure to distinguish between a condition's being true and having various other properties, such as being believed to be true, being known to be true, being represented in a database as true, etc. In OWL-S, if a process's precondition is false, the consequences of performing or initiating the process are undefined.

  The performance of a process may result in changes of the state of the world (effects), and the acquisition of information by the agent performing it (outputs). However, we don't link processes directly to effects and outputs, because process modelers often want to model the dependence of these on context. For example, if a process contains a step to buy an item, there are two possible outcomes: either the purchase succeeds or it fails. In the former case, the effect is that ownership is transferred and the output is, say, a confirmation number. In the latter case, there is no effect, and the output is a failure message.

  We use the term *result* to refer to a coupled output and effect.

```
<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>
```

- **Conditioning Outputs and Effects**

  Having declared a result, a process model can then describe it in terms of four properties

```
<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>
```

The `inCondition` property specifies the condition under which this result (and not another) occurs. The `withOutput` and `hasEffect` properties then state what ensues when the condition is true. The `hasResultVar` property declares variables that are bound in the `inCondition`. These variables, called `ResultVar`s, are analogous to `Local`s, and serve a similar purpose, allowing variables to be bound in preconditions and then used in the declarations regarding outputs and effects.

```
<owl:Class rdf:about="ResultVar">
  <rdfs:subClassOf rdf:resource="#Parameter"/>
  <owl:disjointWith rdf:resource="#Input"/>
  <owl:disjointWith rdf:resource="#Output"/>
  <owl:disjointWith rdf:resource="#Local"/>
</owl:Class>
```

A typical example is a process that charges a credit card. The charge goes through if the card is not overdrawn. If it is overdrawn, the only output is a failure notification. So the description of the process must include the description of two `Result`s, possibly in this form:

```
<process:AtomicProcess rdf:ID="Purchase">
   <process:hasInput/>
      <process:Input rdf:ID="ObjectPurchased"/>
   </process:hasInput>
   <process:hasInput/>
      <process:Input rdf:ID="PurchaseAmt"/>
   </process:hasInput>
   <process:hasInput/>
      <process:Input rdf:ID="CreditCard"/>
   </process:hasInput>
   <process:hasOutput/>
      <process:Output rdf:ID="ConfirmationNum"/>
   </process:hasOutput>
   <process:hasResult>
     <process:Result>
        <process:hasResultVar>
```

```
            <process:ResultVar rdf:ID="CreditLimH">
               <process:parameterType rdf:resource="&ecom;#Dollars"/>
            </process:ResultVar>
         </process:hasResultVar>
         <process:inCondition expressionLanguage="&expr;#KIF"
                         rdf:dataType="&xsd;#string">
            (and (current-value (credit-limit ?CreditCard)
                                ?CreditLimH)
                 (>= ?CreditLimH ?purchaseAmt))
         </process:inCondition>
         <process:withOutput>
            <process:OutputBinding>
               <process:toParam rdf:resource="#ConfirmationNum"/>
               <process:valueFunction rdf:parseType="Literal">
                  <cc:ConfirmationNum xsd:datatype="&xsd;#string"/>
               </process:valueFunction>
            </process:OutputBinding>
         </process:withOutput>
         <process:hasEffect expressionLanguage="&expr;#KIF"
                    rdf:dataType="&xsd;#string">
            (and (confirmed (purchase ?purchaseAmt) ?ConfirmationNum)
                 (own ?objectPurchased)
                 (decrease (credit-limit ?CreditCard)
                           ?purchaseAmt))
         </process:hasEffect>
      </process:Result>
      <process:Result>
         <process:hasResultVar>
            <process:ResultVar rdf:ID="CreditLimL">
               <process:parameterType rdf:resource="&ecom;#Dollars"/>
            </process:ResultVar>
         </process:hasResultVar>
         <process:inCondition expressionLanguage="&expr;#KIF"
                         rdf:dataType="&xsd;#string">
            (and (current-value (credit-limit ?CreditCard)
                                ?CreditLimL)
                 (< ?CreditLimL ?purchaseAmt))
         </process:inCondition>
         <process:withOutput rdf:resource="&ecom;failureNotice"/>
            <process:OutputBinding>
               <process:toParam rdf:resource="#ConfirmationNum"/>
               <process:valueData rdf:parseType="Literal">
                  <drs:Literal>
                     <drs:litdefn xsd:datatype="&xsd;#string">00000000</drs:litdefn>
                  </drs:Literal>
               </process:valueData>
            </process:OutputBinding>
         </process:withOutput>
      </process:Result>
   </process:hasResult>
</process:AtomicProcess>
```

As a result of the execution of the process, a credit card is charged and the money in the account reduced. Note, once again, that there is a fundamental difference between effects

and outputs. Effects describe conditions in the world, while outputs describe information. In a more realistic version of this example, the service may send a notification, or an invoice, that it charged the credit card account. This output is just a datum of one type or another. The effect describes the actual event that the output is part of the description of: that the amount of money in the credit card account has been reduced and that the client now owns the object it intended to purchase.

Finally, there is another output descriptor, called `resultForm`. This is not attached to a variable, but to a Result:

```
<owl:DatatypeProperty rdf:ID="resultForm">
  <rdfs:label>resultForm</rdfs:label>
  <rdfs:domain rdf:resource="#Result"/>
  <rdfs:range rdf:resource="&rdf;#XMLLiteral"/>
</owl:DatatypeProperty>
```

The purpose of `resultForm` is to provide an abstract XML template for outputs sent back to the client. The reason we need such a template are subtle, and do not always apply. Normally the grounding suffices to express how the components of a message are bundled, i.e., how inputs are put together to make a message to a service, and how replies are disassembled into the intended outputs. In essence, all we can or need to do is build up and tear down record structures (in XML Schema terminology, `ComplexType`s). But in the case of a process with multiple Results, it can be extremely useful to specify other features of an output message that indicate which result actually occurred, sparing us the chore of providing output fields to encode that information, or making the client infer it from the form of the other fields. That's what `resultForm` is for.

In our example of a credit-card transaction, we had two Results, one for the case where there was a sufficient balance to pay the bill, and one for when there wasn't. We could augment each result with a further binding, such as this one for the failure case:

```
<owls:Result>
   <owls:hasResultVar>
      <owls:ResultVar rdf:ID="CreditLimL">
         <owls:parameterType rdf:resource="&ecom;#Dollars"/>
      </owls:ResultVar>
   </owls:hasResultVar>
   <owls:inCondition expressionLanguage="&expr;#KIF"
                     rdf:dataType="&xsd;#string">
      (and (current-value (credit-limit ?creditCard)
                          ?CreditLimL)
           (< ?CreditLimL ?purchaseAmt))
   </owls:inCondition>
   <owls:resultForm rdf:parseType="Literal">
       <ecom:CreditExceededFailure>
          <ecom:gap expressionLanguage="&expr;#KIF"
                    rdf:dataType="&xsd;#string">
             (- ?purchaseAmt ?CreditLimL)
          </ecom:gap>
       </ecom:CreditExceededFailure>
   </owls:resultForm>
   <withOutput rdf:resource="&ecom;failureNotice"/>
```

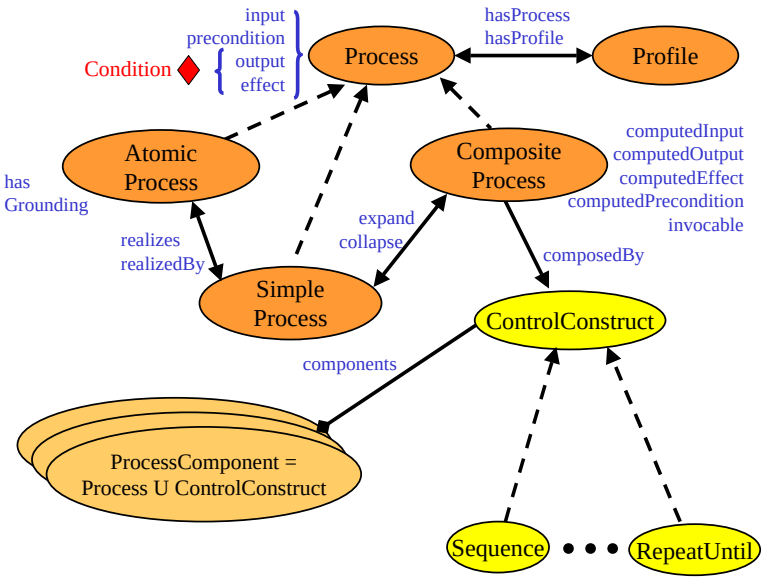

Figure 2: Top level of the process ontology

```
        ...
      </withOutput>
    </owls:Result>
```

5.3 Atomic and Simple Processes

We are now ready to formalize the classes of processes: atomic, composite, and, not mentioned before, "simple."

```
<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>
```

See figure 2.

Atomic processes correspond to the actions a service can perform by engaging it in a single interaction; composite processes correspond to actions that require multi-step protocols; finally, simple processes provide an abstraction mechanism to provide multiple views of the same process. We discuss atomics and simples here, reserving composites for the next subsection.

Atomic processes are directly invocable (by passing them the appropriate messages). Atomic processes have no subprocesses and execute in a single step, as far as the service requester is

concerned. That is, they take an input message, do something, and then return their output message. For each atomic process, there must be provided a grounding that enables a service requester to construct messages to the process and deconstruct replies, as explained in Section 6.

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

{\it Simple} processes are not invocable and are
not associated with a grounding, but, like atomic processes, they {\it
are} conceived of as having single-step executions.  Simple processes
are used as elements of abstraction; a simple process may be used
either to provide a view of (a specialized way of using) some atomic
process, or a simplified representation of some composite process (for
purposes of planning and reasoning).  In the former case, the simple
process is \owlfont{realizedBy} the atomic process; in the latter case, the
simple process \owlfont{expandsTo} the composite process (see below).

\begin{codesize} \begin{verbatim}
<owl:Class rdf:ID="SimpleProcess">
  <rdfs:subClassOf rdf:resource="#Process"/>
  <owl:disjointWith rdf:resource="#AtomicProcess"/>
</owl:Class>

<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>
```

Finally, for an atomic process, there are always only two participants, `TheClient` and `TheServer`:

```
<owl:Class rdf:about="#AtomicProcess">
    <rdfs:subClassOf>
      <owl:Restriction>
<owl:onProperty rdf:resource="#hasClient"/>
<owl:hasValue rdf:resource="#TheClient"/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
<owl:onProperty rdf:resource="#performedBy"/>
<owl:hasValue rdf:resource="#TheServer"/>
      </owl:Restriction>
    </rdfs:subClassOf>

</owl:Class>
```

5.4 Composite Processes

Composite processes are decomposable into other (non-composite or composite) processes; their decomposition can be specified by using control constructs such as `Sequence` and `If-Then-Else`, which are discussed below. Because many of the control constructs have names reminiscent of control structures in programming languages, it is easy to lose sight of a fundamental difference: a composite process is not a behavior a service *will* do, but a behavior (or set of behaviors) the client *can* perform by sending and receiving a series of messages. If the composite process has an overall effect, then the client must perform the entire process in order to achieve that effect. We have not yet given a precise specification of what it means to perform a process, but all we mean is that, e.g., if a composite is a `Sequence`, then the client sends a series of messages that invoke every step in order.

One crucial feature of a composite process is its specification of how its inputs are accepted by particular subprocesses, and how its various outputs are produced by particular subprocesses. We discuss this topic in section 5.5.

```
<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>
```

A `CompositeProcess` must have a `composedOf` property by which is indicated the control structure of the composite, using a `ControlConstruct`.

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

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

Each control construct, in turn, is associated with an additional property called `components` to indicate the nested control constructs from which it is composed, and, in some cases, their ordering.

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

For instance, any instance of the control construct `Sequence` has a `components` property that ranges over a `ControlConstructList` (a list of control constructs). We give a complete table of composite control constructs below.

Any composite process can be considered a tree whose nonterminal nodes are labeled with control constructs, each of which has children specified using `components`. The leaves of the tree are invocations of other processes, indicated as instances of class `Perform`.

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

The `process` property of a perform indicates the process to be performed:

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

When a process is performed as a step in a larger process, there must be a description of where the inputs to the performed process come from and where the outputs go. This issue we defer to section 5.5.

A process can often be viewed at different levels of granularity, either as a primitive, undecomposable process or as a composite process. These are sometimes referred to as "black box" and "glass box" views, respectively. Either perspective may be the more useful in some given context. When a composite process is viewed as a black box, a simple process can be used to represent this. In this case, the relationship between the simple and composite is represented using the `expandsTo` property, and its inverse, the `collapsesTo` property.

```
<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>
```

We conclude this section with an overview of the OWL-S control constructs: Sequence, Split, Split + Join, Choice, Unordered, Condition, If-Then-Else, Iterate, Repeat-While, and Repeat-Until.

Sequence : A list of control constructs to be done in order.

```
<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>
```

Split : The components of a `Split` process are a bag of process components to be executed concurrently. No further specification about waiting or synchronization is made at this level.

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

<owl:Class rdf:ID="ControlConstructBag">
<rdfs:comment> A multiset 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="#ControlConstructBag"/>
   </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Split+Join : Here the process consists of concurrent execution of a bunch of process components with barrier synchronization. With `Split` and `Split+Join`, we can define processes that have partial synchronization (e.g., split all and join some sub-bag).

```
<owl:Class rdf:ID="Split-Join">
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#components"/>
      <owl:allValuesFrom rdf:resource="#ControlConstructBag"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Unordered : Allows the process components (specified as a bag) to be executed in some unspecified order, or concurrently. All components must be executed. As with Split+Join, completion of all components is required. Note that, while the unordered construct itself gives no constraints on the order of execution, nevertheless, in some cases, there may be constraints associated with subcomponents, which must be respected.

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

Examples:

1. If all process components are atomic processes, any ordering is permitted. For instance, (Unordered a b) could result in the execution of a followed by b, or b followed by a.
2. Let a, b, c, and d be atomic processes, and X, Y, and Z be composite processes:

   ```
   X = (Sequence a b)
   Y = (Sequence c d)
   Z = (Unordered X Y)
   ```

   Z, then, translates to the following partial ordering:

   ```
   {(a;b), (c;d)}
   ```

   where ';' means "executes before", and the possible execution sequences (total orders) include

```
{(a;b;c;d), (a;c;b;d), (a;c;d;b), (a;c;d;b),
 (c;d;a;b), (c;a;d;b), (c;a;b;d)}
```

Choice : `Choice` is a control construct whose key property is `chooseFrom`, whose value is a list of processes the execution of one of which constitutes execution of the `Choice`.

[[This wording is a significant scale-down from the original idea, which involved being able to choose an arbitrary number of the processes specified as `chooseFrom`, and then impose further constraints on the set chosen. We have decided to simplify because the machinery used to implement the original idea was not thought through carefully, and because there doesn't seem to be much demand for the complicated version.]]

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

<owl:ObjectProperty rdf:ID="chooseFrom">
   <rdfs:domain rdf:resource="#Choice"/>
   <rdfs:range rdf:resource ="#ControlConstructBag"/>
</owl:ObjectProperty>
```

If-Then-Else : The `If-Then-Else` class is a control construct that has properties `ifCondition`, `then` and *else* holding different aspects of the `If-Then-Else`. Its semantics is intended as "Test `If-condition`; if True do *Then*, if False do `Else`." (Note that the class `Condition`, which is a placeholder for further work, will be defined as a class of logical expressions.)

```
<owl:Class rdf:ID="If-Then-Else">
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#components"/>
      <owl:allValuesFrom rdf:resource="#ControlConstructBag"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty rdf:ID="ifCondition">
 <rdfs:comment> The if condition of an if-then-else</rdfs:comment>
  <rdfs:domain rdf:resource="#If-Then-Else"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="then">
  <rdfs:domain rdf:resource="#If-Then-Else"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="else">
  <rdfs:domain rdf:resource="#If-Then-Else"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>
```

Iterate : The `Iterate` construct makes no assumption about how many iterations are made or when to initiate, terminate, or resume. The initiation, termination or maintenance condition could be specified with a `whileCondition` or an `untilCondition` as below.[3]

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

Repeat-while and Repeat-until Both of these iterate until a condition becomes false or true. Whether the test occurs at a fixed place within the iteration or runs asynchronously varies from subclass to subclass of these classes.

```
<owl:ObjectProperty rdf:ID="whileCondition">
  <rdfs:domain rdf:resource="#Repeat-While"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="whileProcess">
  <rdfs:domain rdf:resource="#Repeat-While"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Repeat-While">
 <rdfs:comment> The repeat while construct</rdfs:comment>
 <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
</owl:Class>


<owl:ObjectProperty rdf:ID="untilCondition">
  <rdfs:domain rdf:resource="#Repeat-Until"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="untilProcess">
  <rdfs:domain rdf:resource="#Repeat-Until"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>
```

[3] Another possible extension is to ability to define counters and use their values as termination conditions. This could be part of an extended process control and execution monitoring ontology.

```
<owl:Class rdf:ID="Repeat-Until">
  <rdfs:comment> The repeat until process</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
</owl:Class>
```

5.5 Specifying Data Flow and Parameter Bindings

When defining processes using OWL-S, there are many places where the input to one process component is obtained as one of the outputs of a preceding step, short-circuiting the normal transmission of data from service to client and back. This is one type of *data flow* from one step of a process to another. There are also other patterns; in particular, the outputs of a composite process may be derived from outputs of some of its components, and specifying which component's output becomes output X of the composite is also a data-flow specification.

We adopt the convention that the source of a datum is identified when the user of the datum is declared. If step 1 feeds step 3, we specify this fact in the description of step 3 rather than the description of step 1. We call this a *consumer-pull* convention, as opposed to a *producer-push* alternative. We implement this convention by providing a notation for arbitrary terms as the values of input or output parameters of a process step, plus a notation for subterms denoting the output or input parameters of prior process steps.

Consider the following tableau:

`I1` input of:{ Composite Process CP: }: with output `O1`

composed of

Step 1: Perform S1 $\longrightarrow$ Step 2: Perform S2

where S1 has inputs `I11` and `I12`, and output `O11`
and S2 has input `I21` and output `O21`

and suppose that we want a straightforward data flow: Input `I1` of the overall process CP is used as input `I11` of S1, after adding 1. Input `I12` of S1 is a constant, the string `"Academic"`. Output `O11` of S1 is used as input `I21` of S2. The maximum of 0 and output `O21` of S2, times π, is used as output `O1` of CP. Using a consumer-pull convention, we simply declare the parameters `I1`, `O11`, and `O21`, but for parameters `I11`, `I21`, and `O1` we provide, in addition to a declaration, *bindings* that specify that

`I11`(S1)	*comes from*	`incr(I1`(CP)`)`
`I12`(S1)	=	`"Academic"`
`I21`(S2)	*comes from*	`O11`(S1)
`O1`(CP)	*comes from*	$\pi \times \max(0, \texttt{O21}(\text{S2}))$

Each of these equalities is represented in OWL-S as a `Binding`, an abstract object with two properties: `toParam`, the name of the parameter (e.g., `I21`(S2)), and `valueSpecifier`, a description of its value. In an effort to provide value specifications in as concise a manner as possible in a variety of situations, we provide three different types: `valueSource`, `valueData`, and `valueFunction`. .

We declare the `toParam` property in the usual way:

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

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

The simplest sort of dataflow spec is `valueSource`:

```
<owl:ObjectProperty rdf:ID="valueSource">
  <rdfs:label>valueSource</rdfs:label>
  <rdfs:domain rdf:resource="#Binding"/>
  <rdfs:range rdf:resource="#ValueOf"/>
  <rdfs:subPropertyOf rdf:resource="#valueSpecifier"/>
</owl:ObjectProperty>
```

The range of `valueSource` is a simple object of class `ValueOf`, specified entirely by its properties `theVar` and `fromProcess`. If a binding with `toParam`$= p$ has `valueSource` $= s$ with properties `theVar`$=v$ and `fromProcess`$= R$, that means that parameter p of this process $=$ parameter v of R.

```
<owl:Class rdf:ID="ValueOf">
  <rdfs:label>ValueOf</rdfs:label>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#theVar"/>
      <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">
                1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#fromProcess"/>
      <owl:maxCardinality rdf:datatype="&xsd;#nonNegativeInteger">
                  1</owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty rdf:ID="theVar">
  <rdfs:domain rdf:resource="#ValueOf"/>
  <rdfs:range rdf:resource="#Parameter"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="fromProcess">
  <rdfs:domain rdf:resource="#ValueOf"/>
```

```
  <rdfs:range rdf:resource="#Perform"/>
</owl:ObjectProperty>
```

For example, our simple tableau has one place where a `ValueSource` expression can be used. Here is how that part of the tableau would be expressed:

```
<process:Sequence rdf:ID="CP">
   <process:components rdf:parseType="Collection">
      <process:Perform rdf:ID="S1">
         <process:process rdf:resource="&aux;#UsefulProcess"/>
         ...
      </process:Perform>
      <process:Perform rdf:ID="S2">
         <process:process rdf:resource="&aux;#AnotherUsefulProcess"/>
         <process:hasDataFrom>   <!-- see below -->
            <process:Binding>
               <process:theParam rdf:resource="&aux;#I21"/>
               <process:valueSource>
                  <process:ValueOf>
                      <process:theParam rdf:resource="#O11"/>
                      <process:fromProcess rdf:resource="#S1"/>
                  </process:ValueOf>
               </process:valueSource>
            </process:Binding>
         </process:hasDataFrom>
      </process:Perform>
   </process:components>
   ...
</process:Sequence>
```

In this example, we used the `hasDataFrom` property, which is used to link inputs of `Performs` to bindings:

```
<owl:ObjectProperty rdf:ID="hasDataFrom">
  <rdfs:domain rdf:resource="#ProcessComponent"/>
  <rdfs:range rdf:resource="#Binding"/>
</owl:ObjectProperty>
```

The complete tableau appears below, after further discussion of bindings.

There is a subtle issue that arises from the fact that the range of `fromProcess` is `Perform`. In our informal tableau above, we used expressions such as `incr(I1`(CP)`)` to mean the the input `I1` of the overall process CP. But we cannot actually refer to a `Binding` with `valueSource` that is a `ValueOf` with `fromProcess` = CP, because CP is not a `Perform`, but a process. If you think about it, it makes no sense to refer to a value extracted from a process, because every time the process is invoked different values will be involved. We need an expression that refers to the "current value" of a parameter, that is, its value during an actual Perform. We want to say: During any Perform P of CP, the value of the input `I1` of step `S1` is the value of the input `I1` of P.

Hence we introduce a standard variable to play the role of P, and give it the name `TheParentPerform`.

```
<swrl:Variable rdf:ID="TheParentPerform">
  <rdfs:comment>
```

```
    A special-purpose variable, used to refer, at runtime, to the execution
    instance of the enclosing composite process definition.
  </rdfs:comment>
</swrl:Variable>
```

We will show how this device is used in our example tableau shortly.

The remaining two data-source specifications use the XML Literal trick to encode arbitrary expressions.

```
<owl:DatatypeProperty rdf:ID="valueFunction">
  <rdfs:label>valueFunction</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#valueSpecifier"/>
  <rdfs:domain rdf:resource="#Binding"/>
  <rdfs:range rdf:resource="&rdf;#XMLLiteral"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="valueData">
  <rdfs:label>valueData</rdfs:label>
  <rdfs:domain rdf:resource="#Binding"/>
</owl:DatatypeProperty>
```

The `valueFunction` of a `Binding` is an XML literal to be read as a functional term. Some of its subterms may be `ValueOf`s specifying outputs of previous terms. As with conditions and effects, the denotation of the `valueFunction` expression cannot be known until variable values are plugged in. The `valueData` of a `Binding` is an XML literal to be interpreted as constant data.

Here is our complete example tableau, with all data flows expressed using one of the three data-source specs.

```
<process:Sequence rdf:ID="CP">
   <process:hasInput>
      <process:Input rdf:ID="I1"/>
   </process:hasInput>
   <process:hasOutput>
      <process:Output rdf:ID="O1"/>
   </process:hasOutput>

   <process:components rdf:parseType="Collection">
      <process:Perform rdf:ID="S1">
         <process:process rdf:resource="&aux;#UsefulProcess"/>
         <process:hasDataFrom>
            <process:InputBinding>
               <process:theParam rdf:resource="&aux;#I11"/>
               <process:valueFunction expressionLanguage="&drs;" rdf:parseType="Literal">
                  <drs:Functional_term>
                     <drs:term_function rdf:resource="&arith;#incr"/>
                     <drs:term_args rdf:parseType="Collection">
                        <process:valueOf>
                            <process:theParam rdf:resource="#I1"/>
                            <process:fromProcess rdf:resource="#TheParentPerform"/>
                        </process:valueOf>
                     </drs:term_args>
```

```
                  </drs:Functional_term>
               </process:valueFunction>
            </process:InputBinding>
            <process:InputBinding>
               <process:theParam rdf:resource="&aux;#I12"/>
               <process:valueData xsd:datatype="&xsd;#string">Academic</process:valueData>
            </process:InputBinding>
         </process:hasDataFrom>
      </process:Perform>
      <process:Perform rdf:ID="S2">
         <process:process rdf:resource="&aux;#AnotherUsefulProcess"/>
         <process:hasDataFrom>
            <process:Binding>
               <process:theParam rdf:resource="&aux;#I21"/>
               <process:valueSource>
                  <process:ValueOf>
                      <process:theParam rdf:resource="#O11"/>
                      <process:fromProcess rdf:resource="#S1"/>
                  </process:ValueOf>
               </process:valueSource>
            </process:Binding>
         </process:hasDataFrom>
      </process:Perform>
   </process:components>

   <process:hasResult>
      <process:withOutput>
         <process:OutputBinding>
             <process:theParam rdf:resource="#O1"/>
             <process:valueFunction expressionLanguage="&drs;" rdf:parseType="Literal">
                <drs:Functional_term>
                    <drs:term_function rdf:resource="&arith;#times"/>
                    <drs:term_args rdf:parseType="Collection">
                       <xsd:Integer rdf:datatype="&xsd;#Integer">1</xsd:Integer>
                       <drs:Functional_term>
                          <drs:term_function rdf:resource="&arith;#max"/>
                          <drs:term_args rdf:parseType="Collection">
                             <xsd:Integer rdf:datatype="&xsd;#Integer">0</xsd:Integer>
                             <process:valueOf>
                                <process:theParam rdf:resource="#O21"/>
                                <process:fromProcess rdf:resource="#S2"/>
                             </process:valueOf>
                          </drs:term_args>
                       </drs:Functional_term>
                    </drs:term_args>
                </drs:Functional_term>
             </process:valueFunction>
         </process:OutputBinding>
      </process:withOutput>
   </process:hasResult>
</process:Sequence>
```

6 Grounding a Service to a Concrete Realization

The grounding of a service specifies the details of how to access the service – details having mainly to do with protocol and message formats, serialization, transport, and addressing. A grounding can be thought of as a *mapping* from an *abstract* to a *concrete* specification of those service description elements that are required for interacting with the service – in particular, for our purposes, the inputs and outputs of atomic processes. Note that in OWL-S, both the *ServiceProfile* and the *ServiceModel* are thought of as abstract representations; only the *ServiceGrounding* deals with the concrete level of specification.

OWL-S does not include an *abstract* construct for explicitly describing messages. Rather, the abstract content of a message is specified, implicitly, by the input or output properties of some atomic process. Thus, atomic processes, in addition to specifying the basic actions from which larger processes are composed, can also be thought of as the communication primitives of an (abstract) process specification.

Concrete messages, however, *are* specified explicitly in a grounding. The central function of an OWL-S grounding is to show how the (abstract) inputs and outputs of an atomic process are to be realized concretely as messages, which carry those inputs and outputs in some specific transmittable format. Due to the existence of a significant body of work in the area of concrete message specification, which is already well along in terms of industry adoption, we have chosen to make use of the Web Services Description Language (WSDL), a particular specification language proposal with strong industry backing, and which we view as representative of such efforts, in crafting an initial grounding mechanism for OWL-S. As mentioned above, our intent here is not to prescribe *the* 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.

Web Services Description Language (WSDL) "is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate" [2].

It may readily be observed that OWL-S' concept of grounding is generally consistent with WSDL's concept of *binding*. Indeed, by using the extensibility elements already provided by WSDL, along with one new extensibility element proposed here, it is an easy matter to ground an OWL-S atomic process. Here, we show how this may be done, relying on the WSDL 1.1 specification.

6.1 Relationships between OWL-S and WSDL

The approach described here allows a service developer, who is going to provide service descriptions for use by potential clients, to take advantage of the complementary strengths of these two specification languages. On the one hand (the abstract side of a service specification), the developer benefits by making use of OWL-S' process model, and the expressiveness of OWL's class typing mechanisms, relative to what XML Schema Definition (XSD) provides. On the other hand (the concrete side), the developer benefits from the opportunity to reuse the extensive work done in WSDL (and related languages such as SOAP), and software support for message exchanges based on these declarations, as defined to date for various protocols and transport mechanisms.

We emphasize that an OWL-S/WSDL grounding involves a *complementary* use of the two

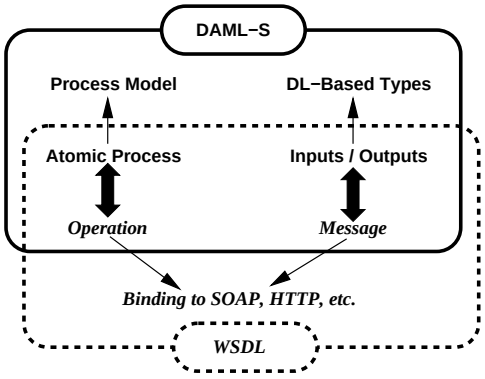

Figure 3: Mapping between OWL-S and WSDL

languages, in a way that is in accord with the intentions of the authors of WSDL. Both languages are required for the full specification of a grounding, because the two languages do not cover the same conceptual space. As indicated by Figure 3, the two languages *do* overlap in the area of providing for the specification of what WSDL calls "abstract types", which in turn are used to characterize the inputs and outputs of services. WSDL, by default, specifies abstract types using XML Schema, whereas OWL-S allows for the definition of abstract types as (description logic-based) OWL classes [4]. However, WSDL/XSD is unable to express the semantics of an OWL class. Similarly, OWL-S has no means, as currently defined, to express the binding information that WSDL captures. Thus, it is natural that a OWL-S/WSDL grounding 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 the following three correspondences between OWL-S and WSDL. Figure 3 shows the first two of these.

1. AN OWL-S atomic process corresponds to a WSDL *operation*. Different types of operations are related to OWL-S processes as follows:

   - 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 WSDL's *solicit-response* operation.[5]

   Note that OWL-S grounding doesn't mandate a one-to-one correspondence between an atomic process and a single WSDL operation (although that is the most normal case).

[4] The data types of XML Schema can also be used in defining OWL properties.

[5] Since a composite process has no grounding, this construct would be grounded indirectly by means of its relationship to a simple process (by the *collapsesTo* property), and hence to an atomic process (by the *realizedBy* property). We are considering whether to create a new kind of atomic process in OWL-S, which corresponds directly to the solicit-response operation.

To accommodate the WSDL-supported practice of providing multiple definitions (within different port types) of the same operation, 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.

2. 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.

3. The types (OWL classes) of the inputs and outputs of an OWL-S atomic process correspond to WSDL's extensible notion of *abstract type* (and, as such, may be used in WSDL specifications of message parts).

To construct an OWL-S/WSDL grounding one must first identify, in WSDL, the messages and operations by which an atomic process may be accessed, and then specify correspondences (1) – (3).

Prior to OWL-S version 0.9, correspondences (2) and (3) were required to be direct correspondences. That is, each OWL-S input or output had to directly match up with a particular WSDL message part; and each input/output type had to literally serve as the type specified in WSDL. Starting with version 0.9, this limitation no longer exists. Version 0.9 allows for the specification of XSLT transformations to show how each WSDL input is derived from (one or more) OWL-S input properties, and how each OWL-S output is derived from (one or more) WSDL output message parts.

Although it is not logically necessary to do so, we believe it will be useful to specify these correspondences both in WSDL and in OWL-S. Thus, as indicated in the following, we allow for constructs in both languages for this purpose.

6.2 Grounding OWL-S Services with WSDL and SOAP

Because OWL-S is an XML-based language, and its atomic process declarations and input and output types already fit nicely with WSDL, it is easy to extend existing WSDL bindings for use with OWL-S, such as the SOAP binding. Here, we indicate briefly how an arbitrary atomic process, specified in OWL-S, can be given a grounding using WSDL and SOAP, with the assumption of HTTP as the chosen transport mechanism.

6.2.1 The *WSDL* Specification

Grounding OWL-S with WSDL and SOAP involves the construction of a WSDL service description with all the usual parts (*types, message, operation, port type, binding*, and *service* constructs).

With respect to the types of the WSDL message parts, it is useful to distinguish two cases: those in which the type is an OWL type (that is, the WSDL service is a "native speaker" of that OWL type); and all others. In the first case, the OWL class can either be defined within the WSDL *types* section, or defined in a separate document and referred to from within the WSDL description, using *owl-s-parameter*, as explained below — in which case its definition can be omitted from the WSDL *types* section.

OWL-S extensions are introduced as follows:

1. In a *part* of the WSDL *message* definition, the *owl-s-parameter* attribute may be used to indicate the fully qualified name of the OWL-S input or output object (instance of class Parameter), to which this part of the message corresponds. This attribute is especially useful for those cases in which the type of the message part is an OWL type (and has not been defined in the WSDL *types* section). In these cases, the OWL type definition can be obtained from the OWL-S process specification, by inspecting the parameterType property of the referenced input or output object.

2. For those cases in which a message part uses an OWL type, the *encodingStyle* attribute, within the WSDL *binding* element, can be given a value such as "http://www.w3.org/2002/07/owl" (or whatever is appropriate for a later version or descendant language of OWL), to indicate that the message parts will be serialized in the normal way for class instances of the given types, for the specified version of OWL.

3. In each WSDL *operation* element, the *owl-s-process* attribute may be used to indicate the name of the OWL-S atomic process, to which the operation corresponds.

Note that WSDL already allows for the use of arbitrary new attributes in message part elements, and for the use of arbitrary values for the *encodingStyle* attribute. Thus, extension (3) above is the only point on which a modification to the current WSDL specification 1.1 is called for.

6.2.2 OWL-S' *Grounding* Class

Thus far, we have only shown how WSDL definitions may refer to the corresponding OWL-S declarations. It remains to establish a mechanism by which the relevant WSDL constructs may be referenced in OWL-S. The OWL-S WsdlGrounding class, a subclass of *Grounding*, serves this purpose. Each WsdlGrounding instance, in turn, contains a list of WsdlAtomicProcessGrounding instances.

A WsdlAtomicProcessGrounding instance refers to specific elements within the WSDL specification, using the following properties:

- *wsdlVersion*: A URI that indicates the version of WSDL in use.
- *wsdlDocument*: A URI of a WSDL document to which this grounding refers.
- *wsdlOperation*: The URI of the WSDL operation corresponding to the given atomic process.
- *wsdlInputMessage*: An object containing the URI of the WSDL message definition that carries the inputs of the given atomic process.
- *wsdlInputs*: An object containing a list of mapping pairs, one for each message part of the WSDL input message. Each such pair is represented using an instance of *WsdlInputMessageMap*. One element of the pair (expressed with the *wsdlMessagePart* property) identifies the message part, using a URI. The other element tells 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.)

- *wsdlOutputMessage*: Similar to *wsdlInputMessage*, but for outputs.
- *wsdlOutputs*: Similar to *wsdlInputs*, but for outputs. In this case, we have a list of mapping pairs, one for each output of the OWL-S atomic process. Each such pair is represented using an instance of WsdlOutputMessageMap, and each pair contains a *owlsParameter* instance specifying the output. The other element of the pair can either be *wsdlMessagePart*, when there is a direct correpondence with a particular message part, or *xsltTransformation* for all other cases.

Additional explanation and examples of how to specify groundings are given in an online document [14].

6.3 Limitations of this Approach

There is an unresolved issue having to do with OWL-S atomic processes that make use of conditional outputs, that is, that specify two or more possible sets of outputs. Because WSDL 1.1 allows only a single output message specification for a given operation, and because OWL-S' treatment of conditional outputs is expected to evolve further, this issue has been left unresolved in the current release (OWL-S 1.1).

7 Resources

Services are effected by processes and processes generally require resources. Therefore, an ontology of resources is an important component of an ontology of services. Our aim here is to propose an ontology of resources stated at an abstract enough level to cover physical, temporal, computational, and other sorts of resources. Specific kinds of resources will, of course, have specific properties; in this development we sketch out the principal classes of properties a resource might have. The OWL-S file *Resource.owl* contains a version of the portions of the ontology that can currently be encoded in OWL. As OWL develops, particularly in the area of expressing rules, the various constraints on concepts in the ontology will be written up in OWL as well.

First of all, a distinction must be pointed out. There are *resource types*, such as fuel. There are *resource tokens*, such as the fuel in the gas tank of a particular car. And there is the quantity, or *capacity*, of the resource token at any given instant, such as the five gallons of fuel in the car's tank right now. We are primarily interested in the second of these notions. Resources in this sense can, depending on resource type, be consumed, replenished, locked, and released. A resource token, or simply *resource*, is what is available to an activity.

7.1 Allocation Types

Resources are allocated to activities or processes. A principal distinction in types of resources concerns their status after the activity stops using them. We will call this the resource's *AllocationType*. If a resource is gone after it is used, its AllocationType is *ConsumableAllocation*. If not, its AllocationType is *ReusableAllocation*.

Examples of reusable resources are the use of a device, the availability of an agent, the use of a region of space, and the use of bandwidth. (These could be viewed as consumable uses of the cross product of the resource (e.g., space) with time, but they are easily decomposed into the resource and time, where only time is consumed.) A persistent resource can be locked and released. When it is locked, it cannot be used by another agent.

Examples of consumable resources are food, charge in a battery, fuel, money, and time. Consumable resources can sometimes be replenished after they are consumed. A deadline is an indirect constraint on the consumable resource of time.

Many resources, such as food, are perishable. We can view this case as having two processes operating on the resource – one functional and relatively rapid, one dysfunctional and relatively slow. Thus, eating food is functional, food spoiling is dysfunctional, and eating is rapid relative to spoiling.

Preconditions on processes can often be viewed as the availability of some resource. Many processes have a location precondition or, more generally, an access precondition. Permission would be an example. In general, if a process is executed as a precondition to another process, we can view its product (or its having been done) as a resource. Something being in the right location for a process's execution can thus be seen as a resource.

7.2 Capacity Types

Resources generally have a precise quantitative measure of capacity at any given instant of time. (Enthusiasm is an interesting limiting case – it is a consumable resource that can be replenished and is required for many tasks, but it cannot be measured precisely. Attention is a similar resource.)

The quantitative measure might be continuous, such as the quantity of fuel. Or it could be a discrete measure, such as a number of chairs occupied. Thus, a resource has a *CapacityType*, where the two CapacityTypes are *DiscreteCapacity* and *ContinuousCapacity*.

Capacity can be related to various other resource-theoretic predicates. In the following rules, for future incorporation into the OWL ontology, R stands for a resource, A for an activity, T for a time interval, and t for a time instant. The expression *use(A,R,T/t)* means that activity A uses resource R over time interval T or for time instant t. The expression *capacity(R,T/t)* refers to the capacity of resource R over time interval T or for time instant t.

The capacity of a persistent resource at the beginning of its use is the same as at the end.

$$reusable(R)\ \&\ use(A,R,T) \Rightarrow capacity(R,start(T)) = capacity(R,end(T))$$

The quantity of a consumable resource at the beginning of its use is more than at the end.

$$consumable(R)\ \&\ use(A,R,T) \Rightarrow capacity(R,start(T)) > capacity(R,end(T))$$

When an agent replenishes a resource during period T, there is more after the replenishment.

$$replenish(A,R,T) \Rightarrow capacity(R,start(T)) < capacity(R,end(T))$$

When a reusable resource is used for period T, it is locked at the beginning of T and released at the end.

$$reusable(R)\ \&\ use(A,R,T) \Rightarrow lock(A,R,start(T))\ \&\ release(A,R,end(T))$$

Capacities of resources can also have a *capacityGranularity*, that is, the units in terms of which the capacity is measured.

7.3 Resource Composition

A resource can be atomic, or it can be an aggregate. Thus, *AtomicResource* and *AggregateResource* are subclasses of *Resource*.

Some atomic resources can be shared by different activities, while others cannot. For example, several activities may need a table but can in fact use the same table. We thus distinguish between unit capacity atomic resources, whose availability to an activity is a yes-no question, and batch capacity atomic resources, which can support multiple activities in a synchronized fashion. *UnitCapacityResource* and *BatchCapacityResource* are subclasses of *AtomicResource*.

Aggregates can be conjunctive or disjunctive. For conjunctive aggregates, all the elements must be allocated to the activity. For a disjunctive aggregate a subset of the elements in the aggregate can be allocated. An example of a disjunctive resource is a process that requires any 3 adjacent chairs of 100 chairs in a room. Thus, *ConjunctiveAggregateResource* and *DisjunctiveAggregateResource* are subclasses of *AggregateResource*.

Shareable resources should be understood in terms of batch capacity resources and aggregation.

A very important use of an ontology of resources could be in a monotonic version of "negation as failure" in a rules language. In this view, "not P" would not be negation as failure. Rather one would use the predicate "cantfind(P,R)" where R is some indication of the resources to devote to the search for a proof of P. For example, R could then be a list or description of Web resources, a certain number of inference steps, or a certain amount of time.

8 Summary and Current Status

OWL-S is an ontology, within the OWL-based framework of the Semantic Web, for describing Web services. It will enable users and software agents to automatically discover, invoke, compose, and monitor Web resources offering services, under specified constraints.

This technical overview accompanies the release of OWL-S version 1.1. The release materials can be found at http://www.daml.org/services/. A variety of other documents on this site give examples, additional documention, and information about limitations of the current release and future directions in the evolution of the ontology.

We expect to enhance OWL-S in the future in ways that we have indicated in this technical overview and elsewhere, and in response to users' experience with it. We believe it will help make the Semantic Web a place where people can not only find information but also get things done.

References

[1] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[2] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315, 2001.

[3] DAML-S Home Page. http://www.daml.org/services/, 2003.

[4] K. Decker, K. Sycara, and M. Williamson. Middle-agents for the Internet. In *IJCAI97*, 1997.

[5] T. Finin, Y. Labrou, and J. Mayfield. KQML as an Agent Communication Language. In J. Bradshaw, editor, *Software Agents*. MIT Press, Cambridge, 1997.

[6] M. Ghallab et al. PDDL-The Planning Domain Definition Language V. 2. Technical Report, report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[7] J. Hendler and D. L. McGuinness. DARPA Agent Markup Language. *IEEE Intelligent Systems*, 15(6):72–73, 2001.

[8] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. Swrl: A semantic web rule language combining owl and ruleml, 2003. Available at `http://www.daml.org/2003/11/swrl/`.

[9] Joint US/EU ad hoc Agent Markup Language Committee. Reference description of the DAML+OIL (March 2001) ontology markup language. http://www.daml.org/2001/03/reference, March 2001.

[10] KIF. Knowledge Interchange Format: Draft proposed American National Standard (dpans). Technical Report 2/98-004, ANS, 1998. Also at `http://logic.stanford.edu/kif/dpans.html`.

[11] G. Klyne and J. J. Carroll. Resource description framework (rdf): concepts and abstract syntax, 2004. W3C Recommendation. Available at `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210`.

[12] O. Lassila. Enabling semantic web programming by integrating Rdf and Common Lisp. In *Proc. First Semantic Web Working Symposium*, 2001. Stanford University.

[13] H. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. GOLOG: A Logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–84, April-June 1997.

[14] D. Martin, M. Burstein, O. Lassila, M. Paolucci, T. Payne, and S. McIlraith. Describing Web Services using OWL-S and WSDL. http://www.daml.org/services/owl-s/1.0/owl-s-wsdl.html, October 2003.

[15] D. Martin, A. Cheyer, and D. Moran. The Open Agent Architecture: A Framework for Building Distributed Software Systems. *Applied Artificial Intelligence*, 13(1-2):92–128, 1999.

[16] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. http://www.w3.org/TR/owl-features/, August 2003. World Wide Web Consortium (W3C) Candidate Recommendation.

[17] S. McIlraith, T. C. Son, and H. Zeng. Mobilizing the Web with DAML-Enabled Web Service. In *Proc. Second Int'l Workshop Semantic Web (SemWeb'2001)*, 2001.

[18] S. McIlraith, T. C. Son, and H. Zeng. Semantic Web Service. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[19] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[20] R. Milner. Communicating with Mobile Agents: The pi-Calculus. Cambridge University Press, Cambridge, 1999.

[21] S. Narayanan. Reasoning About Actions in Narrative Understanding. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI'1999)*, pages 350–357. Morgan Kaufmann Press, San Francisco, 1999.

[22] C. Schlenoff, M. Gruninger, F. Tissot, J. Valois, J. Lubell, and J. Lee. The Process Specification Language (PSL): Overview and Version 1.0 Specification. NISTIR 6459, National Institute of Standards and Technology, Gaithersburg, MD, 2000.

[23] K. Sycara and M. Klusch. Brokering and Matchmaking for Coordination of Agent Societies: A Survey. In A. Omicini et al, editor, *Coordination of Internet Agents*. Springer, 2001.

[24] K. Sycara, M. Klusch, S. Widoff, and J. Lu. Dynamic Service Matchmaking Among Agents in Open Information Environments. *ACM SIGMOD Record (Special Issue on Semantic Interoperability in Global Information Systems)*, 28(1):47–53, 1999.

[25] H.-C. Wong and K. Sycara. A Taxonomy of Middle-agents for the Internet. In *ICMAS'2000*, 2000.