DAML-S: Semantic Markup for Web Services

The DAML 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 DAML-S, that will make 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 DAML-S, and work on related ontologies for resources and for time.

This white paper accompanies DAML-S version 0.9, 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 DAML—the DARPA Agent Markup Language [8], DAML+OIL [9] and OWL [4].[1] 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 and describing services, and the ontology structuring mechanisms of DAML+OIL provide the appropriate framework within which to do this.

*This work is the collaborative effort of projects at BBN Technologies, Carnegie-Mellon University, Nokia, Stanford University, SRI International, and Yale University. Mark Burstein participated for BBN Technologies. The participants for Carnegie-Mellon University are Anupriya Ankolenkar, Massimo Paolucci, Terry Payne, and Katia Sycara. Ora Lassila participated for Nokia. The participants for Stanford University are Sheila McIlraith, Tran Cao Son, and Honglei Zeng. The participants for SRI International are Jerry Hobbs, David Martin, and Srini Narayanan. Drew McDermott participated for Yale.

[1]The current version of DAML-S is built on top of DAML+OIL. Future versions are likely to be built on top of OWL, when it becomes sufficiently mature.

This paper describes a collaborative effort by BBN Technologies, Carnegie Mellon University, Nokia, Stanford University, SRI International, and Yale University, to define just such an ontology. We call this language DAML-S. 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, and related work on ontologies for resources and time, and thoughts toward a future ontology of process control.

This paper accompanies DAML-S version 0.9, which is available at [3]. Please note that, in addition to the DAML+OIL 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 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. DAML-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 sample tasks will give the reader an idea of the kinds of tasks we expect DAML-S to enable [13, 14].

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 DAML-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 DAML-S with a service registry, also called middle agent [5, 21, 12], so that requesters can find it when they query the registry. Thus, DAML-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 of an airline ticket from a particular site 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. DAML-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, DAML-S should 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 DAML-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, DAML-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, DAML-S should provide descriptors for the execution of services. This part of DAML-S is a goal of ours, but it has not yet been defined.

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

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

[2]Service profile has also been called "service capability advertisement" [19].

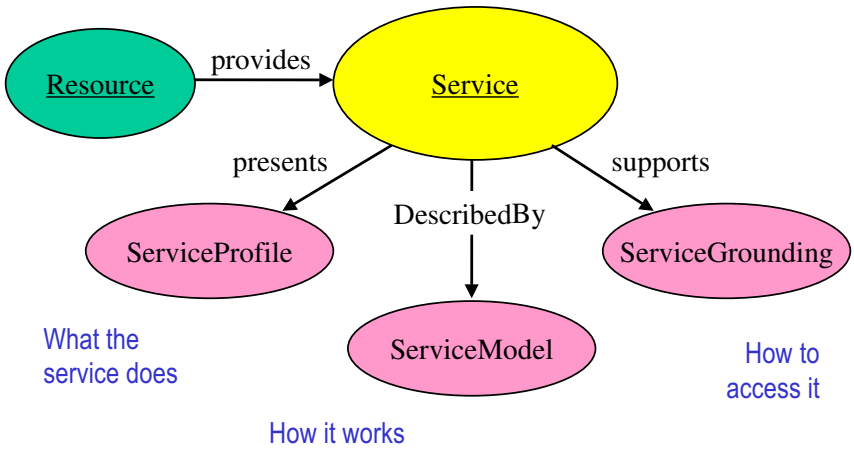

Figure 1: Top level of the service ontology

- *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 a 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, SERVICE-MODEL, 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 service model must be accompanied by at least one supporting grounding. 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 DAML-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 [20, 21]. 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 DAML-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 DAML subclassing it is possible to create specialized representations of services that can be used as service profiles. DAML-S provides one possible representation through the class Profile. A DAML-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 DAML-S Profile provides a few examples of such parameters ranging from an estimate of the max response time, to the geographic radius 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.

DAML-S does not dictate any constraint between Profiles and Process Models, so the two descriptions may be inconsistent without affecting the validity of the DAML 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 form 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 [21, 5]. 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 fields 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 (4.2.4), we discuss the functional representation and specifically the IOPEs; finally, in the last section (4.2.6), 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 specifies a person or other entity that the provider of the service wants to share with the reader. Each item of contact information is an instance of the class Actor described below.

4.2.3 Actor

The class **Actor** provides information on the provider or the requester of the service; specifically, it provides the following information.

name The name property of Actor specifies the name of the actor. This could be either a person name or a company name.

title Title of the contact, a CEO, or ServiceDepartment or whatever is deemed appropriate

phone A phone number that can be used to gather information on the service

fax A fax number that can be used to gather information on the service

email An e-mail address that can be used to gather information on the service

physicalAddress A physical address that can be used to gather information on the service

webURL A URL of the product or company Website

4.2.4 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 DAML-S Profile represents two aspects of the functionality of the service: the information transformation and the state change produced by the execution of the service. 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 information transformation produced by the service is represented by input and output properties of the profile. The input property specifies the information that the service requires to proceed with the computation. For example, a book-selling service could require the credit-card number and bibliographical information of the book to sell. The outputs specify what is the result of the operation of the service. For the book-selling agent the output could be a receipt that acknowledges the sale.

The state change produced by the execution of the service is specified through the precondition and effect properties of the profile. Precondition presents logical conditions that should be

satisfied prior to the service being requested. These conditions should have associated explicit effects that may occur as a result of the service being performed. Effects are the result of the successful execution of a service. The representation of preconditions and effects depends on the representation of rules in the DAML language. Currently, a working group is trying to specify rules in DAML, but no proposal has been put forward. For this reason, the fields precondition and effect are mapped to `thing` meaning that anything is possible, but this will have to be modified in future releases of the profile.

input specifies one of the inputs of the service. It takes as value an instance of ParameterDescription (see below) that specifies an id of the input, a value and a reference to the corresponding input in the process model. The value of the property is an instance of ParameterDescription described below (4.2.5).

output specifies one of the outputs of the service. It takes as value an instance of ParameterDescription (see below) that specifies an id of the output, a value and a reference to the corresponding output in the process model. The value of the property is an instance of ParameterDescription described below (4.2.5).

precondition specifies one of the preconditions of the service. It takes as value an instance of ParameterDescription (4.2.5) that specifies an id of the precondition, a value and a reference to the corresponding precondition in the process model. The value of the property is an instance of ParameterDescription described below (4.2.5).

effect specifies one of the effects of the service. It takes as value an instance of ParameterDescription (4.2.5) that specifies an id of the effect, a value and a reference to the corresponding effect in the process model. The value of the property is an instance of ParameterDescription described below (4.2.5).

4.2.5 ParameterDescription

The class ParameterDescription provides values to inputs and outputs. It collects in one class the name of the input or output that can be used as an identifier, its value and a reference to the corresponding input or output in the process model.

parameterName provides the name of the input or output, which could be just a literal, or perhaps the URI of the process parameter (a property).

restrictedTo provides a restriction on the values of the input or output.

refersTo provides a reference to the input or output in the process model.

4.2.6 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 (4.2.7).

serviceCategory refers to an entry in some ontology or taxonomy of services. The value of the property is an instance of the class ServiceCategory (4.2.9)

QualityRating is used to specify the rating of a service using some rating system. The rating of a service provides the potential client with information about the quality of the service provided. (4.2.8)

4.2.7 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 DAML ontology.

4.2.8 QualityRating

ratingName points to the name of the rating service.

rating stores the value of the rating within a given rating service.

4.2.9 ServiceCategory

ServiceCategory describes categories of services on the bases of some classification that may be outside DAML-S and possibly outside DAML. 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 a service, it can be viewed as a *process*. We have defined a particular subclass of ServiceModel, the ProcessModel, which draws upon well-established work in a variety of fields, such as AI planning and workflow automation, and which we believe supports the representational needs of a very broad array of services on the Web. As with the other DAML-S subontologies, our intent here is not to mandate *the* service modeling 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.

There are two chief components of a process model. The *process* — which describes a service in terms of its inputs, outputs, preconditions, effects, and, where appropriate, its component subprocesses — enables planning, composition and agent/service interoperation. The *process control model* allows agents to monitor the execution of a service request. We refer to the first part as the Process Ontology and the second as the Process Control Ontology. Only the former has

been defined in the current version of DAML-S, but below we briefly describe our intentions with regard to the latter. To support both Process and Process Control specification, we have defined an ontology of resources, and a simple ontology of time, both described below; in subsequent versions these will be elaborated further.

5.1 The Process Ontology

We expect our process ontology to serve as the basis for specifying a wide array of services. In developing the ontology, we draw from a variety of sources, including work in AI on standardizations of planning languages [7], work in programming languages and distributed systems [16, 15], emerging standards in process modeling and workflow technology such as the NIST's Process Specification Language (PSL) [18] and the Workflow Management Coalition effort (http://www.aiim.org/wfmc), work on modeling verb semantics and event structure [17], previous work on action-inspired Web service markup [14], work in AI on modeling complex actions [10], and work in agent communication languages [12, 6].

The primary kind of entity in the Process Ontology is, unsurprisingly, a "process".[3] A process can have any number of inputs, representing the information that is, under some conditions, required for the execution of the process. It can have any number of outputs, the information that the process provides, conditionally, after its execution. Besides inputs and outputs, another important type of parameter specifies the participants in a process. A variety of other parameters may also be declared, including, for physical devices, such things as rates, forces, and knob settings. 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 have conditions associated with them.

More precisely, in DAML-S:

- **Process**

  As shown in Figure 2, we distinguish between three types of processes: *atomic*, *simple*, and *composite*; each of these is described further below.

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

  Class Process has related properties *parameter*, *input*, (conditional) *output*, *participant*, *precondition*, and (conditional) *effect*. *Input*, *output*, and *participant* are categorized as subproperties of *parameter*. The range of each of these properties, at the upper ontology level, is left largely unrestricted. Subclasses of Process for specific domains can use DAML+OIL language elements to indicate more specific range restrictions, as well as cardinality restrictions for each of these properties.

[3] This term was chosen over the terms "event" and "action", in part because it is more suggestive of internal structure than "event" and because it does not necessarily presume an agent executing the process and thus is more general than "action". Ultimately, however, the choice is arbitrary. It is modeled after computational procedures or planning operators.

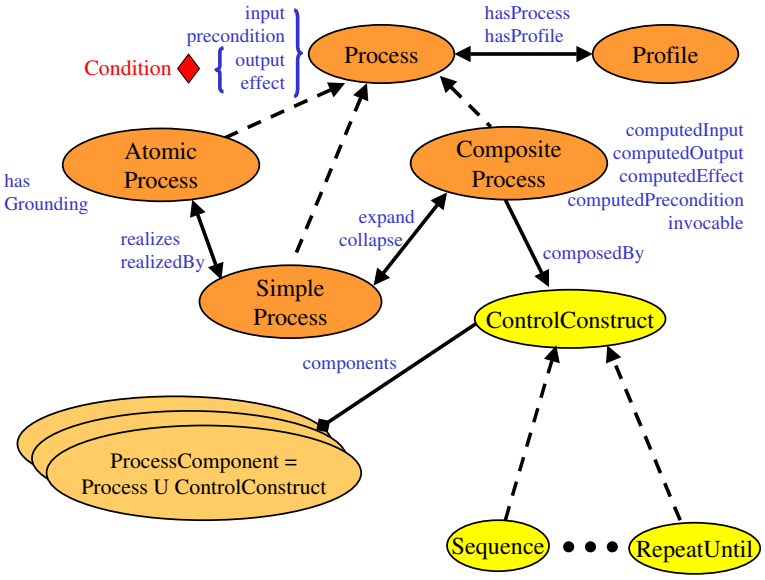

Figure 2: Top level of the process ontology

The following example shows the definition of *parameter*, and its subproperty *input*; the other properties are defined similarly:

```
<rdf:Property rdf:ID="parameter">
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource=""http://www.daml.org/2001/03/daml+oil#Thing"/>
</rdf:Property>

<daml:Property rdf:ID="input">
  <rdfs:subPropertyOf rdf:resource="#parameter"/>
  <daml:range rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
</daml:Property>
```

In addition to its action-related properties, a PROCESS has a number of bookkeeping properties such as *name(rdf:literal)*, *address (URI)*, *documentsRead (URI)*, and *documentsUpdated (URI)*.

- **AtomicProcess**

  The *atomic* processes are directly invocable (by passing them the appropriate messages), have no subprocesses, and execute in a single step, from the perspective of the service requester. That is, they take an input message, execute, and then return their output message — and the service requester has no visibility into the service's execution. For each atomic process, there must be provided a grounding that enables a service requester to construct these messages, as explained in Section 6. But these groundings are normally declared separately from the process model, allowing for easy reuse of process models.

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

- **SimpleProcess**

  Simple processes are not invocable and are not associated with a grounding, but, like atomic processes, they *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 *realizedBy* the atomic process; in the latter case, the simple process *expandsTo* the composite process.

```
<daml:Class rdf:ID="SimpleProcess">
  <daml:subClassOf rdf:resource="#Process"/>
</daml:Class>

<rdf:Property rdf:ID="realizedBy">
  <rdfs:domain rdf:resource="#SimpleProcess"/>
  <rdfs:range rdf:resource="#AtomicProcess"/>
  <daml:inverseOf rdf:resource="#realizes"/>
</rdf:Property>

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

- **CompositeProcess**

  Composite processes are decomposable into other (noncomposite or composite) processes; their decomposition can be specified by using control constructs such as SEQUENCE and IF-THEN-ELSE, which are discussed below. Such a decomposition normally shows, among other things, how the various inputs of the process are accepted by particular subprocesses, and how its various outputs are returned by particular subprocesses.

```
<daml:Class rdf:ID="CompositeProcess">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#composedOf"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>
```

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. The declaration of *expandsTo* is shown above, with SimpleProcess.

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

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

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

Each control construct, in turn, is associated with an additional property called *components* to indicate the ordering and conditional execution of the subprocesses (or control constructs) from which it is composed. For instance, the control construct, Sequence, has a *components* property that ranges over a ProcessComponentList (a list whose items are restricted to be ProcessComponents, which are either processes or control constructs).

```
<rdf:Property rdf:ID="components">
  <rdfs:comment>
    Holds the specific arrangement of subprocesses.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#ControlConstruct"/>
</rdf:Property>

<daml:Class rdf:ID="ProcessComponent">
  <rdfs:comment>
    A ProcessComponent is either a Process or a ControlConstruct.
  </rdfs:comment>
  <daml:unionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Class rdf:about="#ControlConstruct"/>
  </daml:unionOf>
</daml:Class>
```

In the process upper ontology, we have included a minimal set of control constructs that can be specialized to describe a variety of Web services. This minimal set consists of Sequence, Split, Split + Join, Choice, Unordered, Condition, If-Then-Else, Iterate, Repeat-While, and Repeat-Until.

Sequence : A list of Processes to be done in order. We use a DAML+OIL restriction to restrict the components of a Sequence to be a List of process components — which may be either processes (atomic, simple and/or composite) or control constructs.

```
<daml:Class rdf:ID="Sequence">
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <daml:Restriction>
```

```
      <daml:onProperty rdf:resource="#components"/>
      <daml:toClass rdf:resource="#ProcessComponentList"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml: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.

```
<daml:Class rdf:ID="Split">
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#components"/>
      <daml:toClass rdf:resource="#ProcessComponentBag"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

SPLIT is similar to other ontologies' use of Fork, Concurrent, or Parallel. We use the DAML+OIL *sameClassAs* feature to accommodate the different standards for specifying this.

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).

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.

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 A B)
   ```

   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 with additional properties *chosen* and *chooseFrom*. These properties can be used both for process and execution control (e.g., choose from *chooseFrom* and do *chosen* in sequence, or choose from *chooseFrom* and do *chosen*

in parallel) as well for constructing new subclasses like "choose at least n from m", "choose exactly n from m", "choose at most n from m",[4] and so on.

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.)

```
<rdf:Property 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 ="#Condition" </rdfs:range>
</rdf:Property>

<rdf:Property rdf:ID="then">
  <rdfs:domain rdf:resource="#If-Then-Else"/>
  <rdfs:range rdf:resource="#ProcessComponent"/>
</rdf:Property>

<rdf:Property rdf:ID="else">
  <rdfs:domain rdf:resource="#If-Then-Else"/>
  <rdfs:range rdf:resource="#ProcessComponent"/>
</rdf:Property>
```

Iterate : ITERATE is a control construct whose *nextProcessComponent* property has the same value as the current process component. REPEAT is defined as a synonym of the ITERATE class. The repeat/iterate process makes no assumption about how many iterations are made or when to initiate, terminate, or resume. The initiation, termination or maintainance condition could be specified with a *whileCondition* or an *untilCondition* as below.[5]

Repeat-Until : The REPEAT-UNTIL class is similar to the REPEAT-WHILE class in that it specializes the IF-THEN-ELSE class where the *ifCondition* is the same as the *untilCondition* and different from the REPEAT-WHILE class in that the *else* (compared to *then*) property is the repeated process. Thus, the process repeats until the *untilCondition* becomes true.

5.2 Specifying Data Flow; Parameter Bindings

When defining processes using DAML-S, there are many places where different properties of a process, or elements referred to by process properties, should be equated, in the sense that the information denoted by the objects of these properties should be the identical whenever the process is instantiated. A simple example is an atomic process to buy something, where the item to be purchased is referred to by some name or identifier provided as an input to the process, and the various process outputs refer to the same identifier, perhaps as parts of a message saying whether the transaction succeeded or failed. There are many places where this equivalence needs to be stated for the process model to be successfully applied by an agent, including:

[4] This can be obtained by restricting the size of the Process Bag that corresponds to the *components* of the chosen and chooseFrom subprocesses using cardinality, min-cardinality, max-cardinality to get choose(n, m)($0 \leq n \leq |components(chooseFrom)|, 0 < m \leq |components(chosen)|$).

[5] 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.

- In relating process inputs to the process' conditions, outputs or effects, including its preconditions, the conditions governing conditional effects and conditional outputs, and (properties of) the effects and outputs themselves.
- In relating the inputs and outputs of a composite process to the inputs and outputs of its various component subprocesses.
- In relating the inputs and outputs of elements of a composite process definition to parameters of other process components. For example, when a composite process is defined as a sequence of subprocesses, the output of one component of the sequence may well be an input to a subsequent component of the sequence.

In a programming language or in a logic language, we would show how these elements were related using variables. In programming, the variables would be function arguments or local variables. They would be referenced in a function body, to indicate how, for example, an argument of some step was the same as an input to the whole function, and how it came from the output of a previous step.

DAML+OIL does not provide for the use of variables, especially when defining related classes in an ontology. There is no way to state in a class definition that one of the class properties is referenced elsewhere by a variable name, and that this indicates that the properties' values will be identical when the structure is instantiated. Using DAML+OIL, one can only define the inputs and outputs of processes as properties with range restrictions representing the /em classes of allowed values, independent of any context.

We have considered many schemes to address this limitation in DAML+OIL expressivity, and have, for this release, adopted the following DAML+OIL notation. The intent is to capture, purely as a set of process annotations, this critical information about how processes are to be instantiated and information shared between process elements. We have extended our process ontology with the classes and properties used in this notation. The use of this notation in a process definition will enable a specialized DAML-S process reasoner to use this information to determine which properties should have "the same value" in any coherent instance of the process being defined.

In this notation, an instance of the class ValueOf, with properties *atClass* and *theProperty* denotes the object (value) of the specified property at the specified class. This style of reference is intended to be used only within the context of a process being annotated using the property *sameValues*, which relates a process class to a collection of ValueOf objects. The set of referenced *ValueOf* elements are considered to share the same information, as if their values were represented by a single variable.

The DAML+OIL definitions of these properties are as follows:

```
<!-- Used to annotate a process component by describing which
     properties share values.
     The range is a List of ValueOf instances.  -->
<rdf:Property rdf:ID="sameValues">
  <daml:domain rdf:resource="#ProcessComponent"/>
  <daml:range rdf:resource="http://www.daml.org/2001/03/daml+oil#List"/>
</rdf:Property>

<daml:Class rdf:ID="ValueOf"/>
```

```
<!-- This property indicates the class (usually a Process) having the
     referenced property -->
<rdf:Property rdf:ID="atClass">
  <rdfs:domain rdf:resource="#ValueOf"/>
  <rdfs:range  rdf:resource="http://www.daml.org/2001/03/daml+oil#Class"/>
</rdf:Property>

<!-- The property (usually a parameter) whose values are
     referred to. -->
<rdf:Property rdf:ID="theProperty">
  <rdfs:domain rdf:resource="#ValueOf"/>
  <rdfs:range  rdf:resource="http://www.daml.org/2001/03/daml+oil#Property"/>
</rdf:Property>
```

5.3 Process Control Ontology

A process instantiation represents a complex process that is executing in the world. To monitor and control the execution of a process, an agent needs a model to interpret process instantiations with three characteristics:

1. It should provide the mapping rules for the various input state properties (inputs, preconditions) to the corresponding output state properties.
2. It should provide a model of the temporal or state dependencies described by constructs such as sequence, split, split+join, and so forth.
3. It should provide representations for messages about the execution state of atomic and composite processes sufficient to do execution monitoring. This allows an agent to keep track of the status of executions, including successful, failed and interrupted processes, and to respond to each appropriately.

We have not defined a process control ontology in the current version of DAML-S, but we plan to in a future version.

5.4 Time

In conjunction with DAML-S, a temporal ontology is being developed. It is intended to cover a wide range of temporal constructs, including topological relations among instants and intervals, notions of duration, the clock and calendar, and a treatment of temporal granularity. The most recent write-up on the ontology can be found at
http://www.ai.sri.com/daml/ontologies/time/Time.text.

A subset of this ontology has been coded in DAML+OIL – essentially, that part of the ontology that is naturally expressible in description logic. This can be found at
http://www.ai.sri.com/daml/ontologies/time/Time.daml.

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 DAML-S, both the *ServiceProfile* and the *ServiceModel* are thought of as abstract representations; only the *ServiceGrounding* deals with the concrete level of specification.

DAML-S does not include an *abstract* construct for 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 a DAML-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 DAML-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 DAML-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 a DAML-S atomic process. Here, we show how this may be done, relying on the WSDL 1.1 specification.

6.1 Relationships between DAML-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 DAML-S' process model, and the expressiveness of DAML+OIL's class typing mechanisms, relative to what XML Schema 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 a DAML-S/WSDL grounding involves a *complementary* use of the two 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

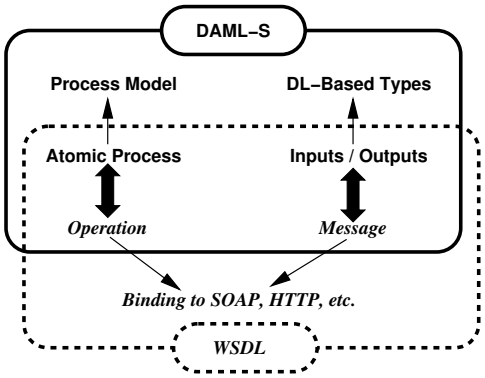

Figure 3: Mapping between DAML-S and WSDL

using XML Schema, whereas DAML-S allows for the definition of abstract types as (description logic-based) DAML+OIL classes [6]. However, WSDL/XSD is unable to express the semantics of a DAML+OIL class. Similarly, DAML-S has no means, as currently defined, to express the binding information that WSDL captures. Thus, it is natural that a DAML-S/WSDL grounding uses DAML+OIL 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.

A DAML-S/WSDL grounding is based upon the following three correspondences between DAML-S and WSDL. Figure 3 shows the first two of these.

1. A DAML-S atomic process corresponds to a WSDL *operation*. Different types of operations are related to DAML-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.[7]

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

[6] The data types of XML Schema can also be used in defining DAML+OIL properties.

[7] 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 DAML-S, which corresponds directly to the solicit-response operation.

2. The set of inputs and the set of outputs of a DAML-S atomic process each correspond to WSDL's concept of *message*. More precisely, DAML-S inputs correspond to the parts of an input message of a WSDL operation, and DAML-S outputs correspond to the parts of an output message of a WSDL operation.

3. The types (DAML+OIL classes) of the inputs and outputs of a DAML-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 a DAML-S/WSDL grounding one must first identify, in WSDL, the messages and operations by which an atomic process may be accessed, and then, to specify correspondences (1) – (3).

Prior to DAML-S version 0.9, correspondences (2) and (3) were required to be direct correspondences. That is, each DAML-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) DAML-S input properties, and how each DAML-S output property 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 DAML-S. Thus, as indicated in the following, we allow for constructs in both languages for this purpose.

6.2 Grounding DAML-S Services with WSDL and SOAP

Because DAML-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 DAML-S, such as the SOAP binding. Here, we indicate briefly how an arbitrary atomic process, specified in DAML-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 DAML-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 a DAML+OIL type (that is, the WSDL service is a "native speaker" of that DAML+OIL type); and all others. In the first case, the DAML+OIL 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 *daml-s-parameter*, as explained below — in which case its definition can be omitted from the WSDL *types* section.

DAML-S extensions are introduced as follows:

1. In a *part* of the WSDL *message* definition, the *daml-s-parameter* attribute may be used to indicate the fully qualified name of the DAML-S input or output property(ies), 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 a DAML+OIL type (and has not been defined in the WSDL *types* section). In these cases, the DAML+OIL type definition can be obtained, from the DAML-S process specification, by inspecting the range of the indicated DAML-S property.

2. For those cases in which a message part uses a DAML+OIL type, the *encodingStyle* attribute, within the WSDL *binding* element, can be given a value such as "http://www.daml.org/2001/03/daml+oil.daml" (or whatever is appropriate for a later version or descendant language of DAML+OIL), 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 DAML.

3. In each WSDL *operation* element, the *daml-s-process* attribute may be used to indicate the name of the DAML-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 DAML-S' *Grounding* Class

Thus far, we have only shown how WSDL definitions may refer to the corresponding DAML-S declarations. It remains to establish a mechanism by which the relevant WSDL constructs may be referenced in DAML-S. The DAML-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 DAML-S atomic process. In the simplest cases — in which the message part corresponds directly to a single DAML-S input property, and the type of the property is directly used by the WSDL specification — this is done just by mentioning the URI of a particular input property (using the *damlsParameter* 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 property of the DAML-S atomic process. Each such pair is represented using an instance of WsdlOutputMessageMap, and each pair contains a *damlsParameter* instance specifying the output property. 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 [11].

6.3 Limitations of this Approach

There is an unresolved issue having to do with DAML-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 DAML-S' treatment of conditional outputs is expected to evolve further, this issue has been left unresolved in the current release (DAML-S 0.9).

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 DAML-S file *Resource.daml* contains a version of the portions of the ontology that can currently be encoded in DAML+OIL. As DAML develops, particularly in the area of expressing rules, the various constraints on concepts in the ontology will be written up in DAML 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 DAML 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 DAML-L. 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

DAML-S is an attempt to provide an ontology, within the framework of the DARPA Agent Markup Language, 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 a fourth prerelease of DAML-S, version 0.9. The prerelease materials can be found at http://www.daml.org/services/. The *Status* page on this site gives additional information about limitations and incomplete aspects of the current release, and directions leading toward a version 1.0.

We expect to enhance DAML-S in the future in ways that we have indicated in this white paper, 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/, 2002.

[4] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language 1.0 Reference. http://www.w3.org/TR/owl-ref/, July 2002.

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

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

[7] 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.

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

[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] 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.

[11] D. Martin, M. Burstein, O. Lassila, M. Paolucci, T. Payne, and S. McIlraith. Describing Web Services using DAML-S and WSDL. http://www.daml.org/services/daml-s/0.7/daml-s-wsdl.html, August 2002.

[12] 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.

[13] 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.

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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