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 have begun to develop 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, the service profile for advertising services, and the process model for the detailed description of the operation of services.

This white paper accompanies DAML-S version 0.6, which is available at http://www.daml.org/services/daml-s/2001/10/.

1 Introduction: Services on the Semantic Web

Efforts toward the creation of the Semantic Web are gaining momentum [2]. 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 DAML—the DARPA Agent Markup Language [7]. DAML enables 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 DAML, 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 provide the appropriate framework within which to do this.

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

*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.

the ontologies for profiles, processes, and time, and thoughts toward a future ontology of process control. We then compare DAML-S with a number of recent industrial efforts to standardize a markup language for services.

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 [10, 11].

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 [3, 18, 9], 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. Users may want to know during this period what the status of their request is, or their plans may have changed requiring alterations in the actions the software agent takes. For example, users may want to make sure their 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

The class SERVICE stands at the top of a taxonomy of services, and its properties are the properties normally associated with all kinds of services. The upper ontology for services is silent as to what the particular subclasses of SERVICE should be, or even the conceptual basis for structuring this taxonomy, but it is expected that the taxonomy will be structured according to functional and domain differences and market needs. For example, one might imagine a broad subclass, B2C-TRANSACTION, which would encompass services for purchasing items from retail Web sites, tracking purchase status, establishing and maintaining accounts with the sites, and so on.

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[1]." 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
- *How is it used?* The answer to this question is given in the "grounding." Thus, the class SERVICE *supports* a SERVICEGROUNDING

[1] A service profile has also been called service capability advertisement [16].

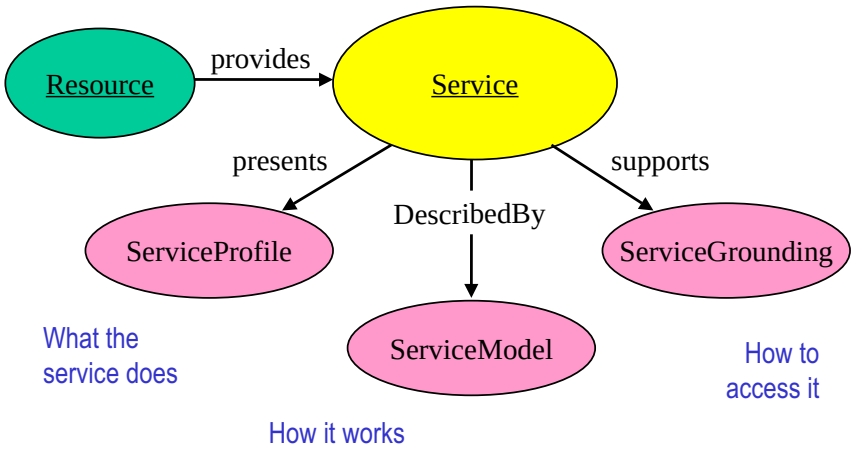

Figure 1: Top level of the service ontology

The properties *presents*, *describedBy*, and *supports* are properties of SERVICE. The classes SERVICEPROFILE, SERVICEMODEL, and SERVICEGROUNDING are the respective ranges of those properties. We expect that each descendant class of SERVICE, such as B2C-TRANSACTION, 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 type of information needed by a service-seeking agent to determine whether the service meets its needs (typically such things as input and output types, preconditions and postconditions, and binding patterns). In future versions, we will use logical rules or their equivalent in such a specification for expressing interactions among parameters. For instance, a rule might say that if a particular input argument is bound in a certain way, certain other input arguments may not be needed, or may be provided by the service itself. As DAML and DAML-S and their applications evolve, logical rules and inferential approaches enabled by them are likely to play an increasingly important role in models and groundings, as well as in profiles. See [4] for additional examples.

The service model tells "how the service works"; that is, it describes what happens when the service is carried out. For non-trivial 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; (4) to monitor the execution

of the service. For non-trivial services, the first two tasks require a model of action and process, the last two involve, in addition, an execution model.

A service grounding ("grounding" for short) specifies the details of how an agent can access a service. Typically a grounding will specify a communications protocol (e.g., RPC, HTTP-FORM, CORBA IDL, SOAP, Java RMI, OAA ACL [9]), and 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 marshaling/serialization techniques employed). The likelihood is that a relatively small set of groundings will come to be widely used in conjunction with DAML services. If this turns out to be the case, groundings will likely be specified at various well-known URIs.

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 deliberately does not specify any cardinalities for the properties *presents*, *describedBy*, and *supports*. Although, in principle, a service needs all three properties to be fully characterized, it is possible to imagine situations in which a partial characterization could be useful. Hence, there is no specification of a minimum cardinality. Further, it should certainly be possible for a service to offer multiple profiles, multiple models, and/or multiple groundings. Hence, there is no specification of a maximum cardinality.

In general, there need not exist a one-to-one correspondence between profiles, models, and/or groundings. The only constraint among these three characterizations that might appropriately be expressed at the upper level ontology is that for each model, there must be at least one supporting grounding.

In the following two sections we discuss the service profile and the service model in greater detail (Service groundings are not discussed further, but will be covered in greater depth in a subsequent publication.)

4 Service Profiles

In a market place of web services, we can broadly identify three classes of entities: service providers, service requesters and infrastructure components, most likely registries, that map the needs of the requesters with the offers of the providers [17, 18]. 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 this framework, service profiles provide a way to describe the services offered by the providers, and the services needed by the requesters.

A service profile contains three types of information: a human readable description of the service and its provider; a specification of the functionalities that are provided by the service; and a host of attributes which provide additional information and requirements such as quality guarantees, expected response time and geographic constraints, these attributes and assist reasoning about several services with similar capabilities. The functionalities of the service are represented by the inputs and preconditions required by the service to the outputs and effects produced. For example, a news reporting service would advertise itself as a service that, given a date, will return the news reported on that date.

A service profile describes who provides the service, the expected quality of the service and the transformation produced by the service in terms of what it expects to run correctly, and what

results it produces. Specifically, the service profile specifies the preconditions that have to be satisfied to use the service effectively, and the inputs that the service expects; furthermore, the service profile specifies the expected effects that result from the execution of the service and the output information returned. The input, outputs, precondition and effects (hereafter IOPEs) of the profile are correspond to the IOPEs of the process model that it advertises. Currently, due to limitations of the DAML language, there is no logical relationship between the service profile IOPEs and the actual IOPEs of the corresponding process model. Therefore, at least in theory, the two descriptions may be inconsistent. Nevertheless, the intended use of these descriptions is to correctly characterize the key parameters of the process model so that potential clients can properly find offered services that meet their needs. A violation of the IOPEs intended use results in a misrepresentation of the service in the registries, therefore service cannot be selected when requested, or it is selected to satisfy a request that it cannot satisfy, resulting in both cases in a loss for the misrepresented service.

Implicitly, the service profiles specify the intended purpose of the service, because they specify only those functionalities that are publicly provided. A book-selling service may involve two different functionalities: it allows other services to browse its site to find books of interest, and eventually to buy the books they found. The book-seller has the choice of advertising just the book-buying service or both the browsing functionality and the buying functionality. In the latter case the service makes public 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 that do not intend to buy. The decision as to which functionalities to advertise determines how the service will be used: a requester that 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. While this is the model followed by web registries such as UDDI, other form of registries are also possible. For example, when the demand of a service is higher then 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. Indeed the type of registries may vary widely and as many as 28 different types have been identified [18, 3]. 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 registries.

Depending on the discovery mechanism followed, the Profile class may need to be modified to satisfy accommodate information. This is possible in DAML by defining specialized subclasses of Profile. For example, when passive registration is used so that services do not advertise directly, rather wait for a web crawler to find what they offer or what they need, the distinction between a request of service and an offer of service is facilitated by the definition of subclasses of profile, as below:

```
<rdfs:Class rdf:ID="OfferedService">
  <rdfs:label>OfferedService</rdfs:label>
  <rdfs:subClassOf rdf:resource="&service;#ServiceProfile" />
</rdfs:Class>
```

Any instance of the class "OfferedService" would be understood as an advertisement of a new service. A similar class for "NeededServices" would instead be used to advertise needs of services.

4.1 Profile Properties

In the following we describe in detail the fields of the profile model; we classify them into three broad categories: the first one, that we name Provenance and Description, describes the provider of the service, an natural language description and additional information that allow the location of the service.

4.1.1 Provenance and Description

Information about the service, such as its provenance, a text summary etc is provided within the profile. This is primarily for use by human users, although these properties are considered when locating requested services.

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

intendedPurpose The property IntendedPurpose provides information on what constitutes successful accomplishment of a service execution.

textDescription The property textDescription provides a brief description of the service. It summarizes what the service offers, or to describe what service is requested.

providedBy The property providedBy links the service profile to an Actor who provides the service.

requestedBy The property providedBy links the service profile to an Actor who requests the service.

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.

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

4.1.2 Functionality Description

An essential component of the profile is the specification of what the service provides and the specification of the conditions that have to 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 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.

```
<rdf:Property rdf:ID="input">
  <rdfs:comment>
    Property describing the inputs of a service in the Service Profile
  </rdfs:comment>
  <rdfs:domain rdf:resource="#ServiceProfile"/>
  <rdfs:subPropertyOf rdf:resource="#ParameterDescription"/>
</rdf:Property>
```

While inputs and outputs represent the service, they are not the only things affected by the operations of the service. For example, to complete the sale the book-selling service requires that the credit card is valid and not overdrawn or expired. In addition, the result of the sale is not only that the buyer owns the book (as specified by the outputs), but that the book is physically transferred from the the warehouse of the seller to the house of the buyer. These conditions are specified by precondition and effect properties of the profile. Precondition present a logical condition 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 events that are caused by the successful execution of a service. The representation of preconditions and effects depends on the representation of rules in the DAML language. Currently, there is a working group that is trying to formulate 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.

```
<rdf:Property rdf:ID="precondition">
  <rdfs:domain rdf:resource="#ServiceProfile"/>
  <rdfs:range rdf:resource="#ConditionDescription"/>
</rdf:Property>
```

Finally, the Profile allows the specification of what domainResources are affected by the use of the service. These domain resources may include computational resources such as bandwidth or disk space as well as more material resources consumed when the service controls some machinery. This type of resource may include fuel, or materials modified by the machine.

The properties of Service Profiles are described in detail below.

input The property 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.

output The property 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.

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

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

domainResource DomainResource(s) - not to be confused with RDF resources, or domain restrictions - specifies resources that are necessary for the task to be executed. No range restrictions are placed on them at present (as with those used by the process model). Specific service descriptions will specialize this property by restricting the range appropriately using subPropertyOf.

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

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

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

The class **ConditionDescription** is used to provide a condition to preconditions and effects. It collects in one class the name of a precondition or an effect that can be used as an identifier, its value and a reference to the corresponding precondition or effect in the process model.

conditionName The property conditionName provides the name of the precondition or effect , which could be just a literal, or perhaps the URI of the process parameter (a property)

statement The property statement provides room for a logical statement that specifies the precondition or the effect. At the present time, the DAML language does not provide any way to express such constraints therefore the value of statement is a daml:Thing (anything goes). This will have to change in future releases as soon as a DAML rule language will be specified.

refersTo The property restrictedTo provides a reference to the precondition or effect in the process model.

4.1.3 Functional Attributes

In the previous section we introduced the functional description of services, yet there are other aspects of services that the users should be aware of. Whilst a service may be accessed from anywhere on the Internet, it may only be applicable to a specific audience. For instance, although it is possible to order food for delivery from a Pittsburgh-based restaurant website in general, one cannot reasonably expect to do this from California. Functional attributes address the problem that there are other properties that can be used to describe a service other than a functional process. These properties are described below.

geographicRadius The geographic radius refers to the geographic scope of the service. This may be at the global or national scale (e.g. for e-commerce) or at a local scale (eg pizza delivery).

degreeOfQuality This property provide qualifications about the service. For example, the following two sub-properties are examples of different degrees of quality, and could be defined within some additional ontology.

serviceParameter An expandable list of properties that may accompany a profile description.

communicationThru This property provides a high-level summary of how a service may communicate, such as what agent communication language (ACL) is used (eg FIPA, KQML, SOAP etc). This summarizes the descriptions provided by the service grounding and are used when matching services; but is not intended to replace the detail provided by the service grounding.

serviceType The service type refers to a high level classification of the service, for example B2B, B2C etc.

serviceCategory The service category refers to an ontology of services that may be on offer. High level services could include Products as well as Problem Solving Capabilities, Commercial Services, Information and so on.

qualityGuarentees These are guarantees that the service promises to deliver, such as guaranteeing to provide the lowest possible interest rate, or a response within 3 minutes, etc.

qualityRating The quality rating property represents an expandable list of rating properties that may accompany a service profile. These ratings refer to industry accepted ratings, such as the Dun and Bradstreet Rating for businesses, or the Star rating for Hotels. For example:

```
<!-- Dun and Bradstreet Rating -->
<rdf:Property rdf:ID="dAndBRating">
  <rdfs:subPropertyOf rdf:resource="#qualityRating" />
</rdf:Property>
```

As a result of the service profile, the user, be it a human, a program or another service, would be able to identify what the service provides, what conditions result from the service and whether the service is available, accessible and how it compares with other functionally equivalent services.

5 Modeling Services as Processes

To give a more 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 will support the representational needs of a very broad array of services on the Web.

The two chief components of a process model are the *process* — which describes a service in terms of its inputs, outputs, preconditions, effects, and, where appropriate, its component subprocesses — and enables planning, composition and agent/service interoperation; and the *process control model*, which allows agents to monitor the execution of a service request. We will 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 drew from a variety of sources, including work in AI on standardizations of planning languages [6], work in programming languages and distributed systems [13, 12], emerging standards in process modeling and workflow technology such as the NIST's Process Specification Language (PSL) [15] and the Workflow Management Coalition effort (http://www.aiim.org/wfmc), work on modeling verb semantics and event structure [14], previous work on action-inspired Web service markup [11], work in AI on modeling complex actions [8], and work in agent communication languages [9, 5].

The primary kind of entity in the Process Ontology is, unsurprisingly, a "process".[2] 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**

```
<rdfs:Class rdf:ID="Process">
  <rdfs:comment> Top-level class for describing how a service works
  </rdfs:comment>
</rdfs: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 THING; that is, left totally unrestricted. Subclasses of PROCESS for specific domains can use DAML language elements to indicate more specific range restrictions, as well as cardinality restrictions for each of these properties.

The following example shows the definition of *parameter*; 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>
```

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

In DAML-S, as shown in figure 2, we distinguish between three types of processes: *atomic*, *simple*, and *composite*.

[2] 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.

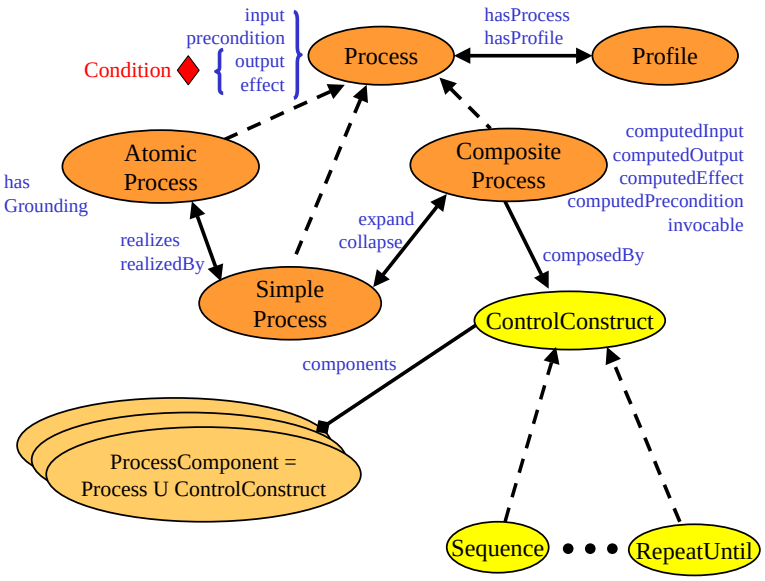

Figure 2: Top level of process ontology

- **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 input message(s) all at once, execute, and then return their output message(s) all at once. Atomic processes must provide a grounding that enables a service requester to construct these messages.

```
<daml:Class rdf:ID="AtomicProcess">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Restriction daml:minCardinality="1">
      <daml:onProperty rdf:resource="#hasGrounding"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>

<rdf:Property rdf:ID="hasGrounding">
  <rdfs:domain rdf:resource="#AtomicProcess"/>
  <rdfs:range rdf:resource="grounding:Grounding"/>
</rdf:Property>
```

- **SimpleProcess**

  Simple processes, on the other hand, 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 *expand*s to 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="expand">
  <rdfs:domain rdf:resource="#SimpleProcess"/>
  <rdfs:range rdf:resource="#CompositeProcess"/>
  <daml:inverseOf rdf:resource="#collapse"/>
</rdf:Property>
```

- **CompositeProcess**

  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. 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:minCardinality="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 *expand* property, and its inverse, the *collapse* property. The declaration of *expand* 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 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.

```
<rdfs:Class rdf:ID="Sequence">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <rdfs:Class> rdf:about="#ControlConstruct" </rdfs:Class>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#components"/>
      <daml:toClass rdf:resource="#ProcessComponentList"/>
    </daml:Restriction>
  <daml:intersectionOf>
</rdfs: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.

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

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

Unordered : Here a bag of process components can be executed in any order. No further constraints are specified. All process components must be executed.

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).

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" [3], 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 place-holder 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

[3] 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)($0 \leq n \leq |components(chooseFrom)|, 0 < m \leq |components(chosen)|$).

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.[4]

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 till the *untilCondition* becomes true.

5.2 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 the sequence, split, split+join, etc constructs.
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.

5.3 Time

For the initial version of DAML-S we have defined a very simple upper ontology for time. There are two classes of entities—INSTANTS and INTERVALS. Each is a subclass of TEMPORAL-ENTITY.

There are three relations that may obtain between an instant and an interval, defined as DAML-S properties:

1. The *start-of* property whose domain is the Interval class and whose range is an Instant.
2. The *end-of* property whose domain is the Interval class and whose range is an Instant.
3. The *inside* property whose domain is the Interval class and whose range is an Instant.

No assumption is made that intervals *consist of* instants.

There are two possible relations that may obtain between a process and one of the temporal objects. A process may be in an *at-time* relation to an instant or in a *during* relation to an interval. Whether a particular process is viewed as instantaneous or as occuring over an interval is a granularity decision that may vary according to the context of use. These relations are defined in DAML-S as properties of processes.

[4] 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.

1. The *at-time* property: its domain is the Process class and its range is an Instant.

2. The *during* property: its domain is the Process class and its range is an Interval.

Viewed as intervals, processes could have properties such as startTime and endTime which are synonymous (daml:samePropertyAs) with the Start-Of and End-Of relation that obtains between intervals and instants.

One further relation can hold between two temporal entities—the *before* relation. The intended semantics is that for an instant or interval to be before another instant or interval, there can be no overlap or abutment between the former and the latter. In DAML-S the *before* property whose domain is the Temporal-entity class and whose range is a Temporal-entity.

Different communities have different ways of representing the times and durations of states and events (processes). For example, states and events can both have durations, and at least events can be instantaneous; or events can only be instantaneous and only states can have durations. Events that one might consider as having duration (e.g., heating water) are modeled as a state of the system that is initiated and terminated by instantaneous events. That is, there is the instantaneous event of the start of the heating at the start of an interval, that transitions the system into a state in which the water is heating. The state continues until another instantaneous event occurs—the stopping of the event at the end of the interval. These two perspectives on events are straightforwardly interdefinable in terms of the ontology we have provided. Thus, DAML-S supports both.

The various relations between intervals defined in Allen's temporal interval calculus [1] can be defined in a straightforward fashion in terms of *before* and identity on the start and end points. For example, two intervals meet when the end of one is identical to the start of the other. Thus, in the near future, when DAML is augmented with the capability of defining logical rules, it will be easy to incorporate the interval calculus into DAML-S. In addition, in future versions of DAML-S we will define primitives for measuring durations and for specifying clock and calendar time.

6 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.

6.1 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 *Allo-*

cationType. 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.

6.2 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 an 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.

6.3 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. R could then be a list or description of web resources, a certain number of inference steps, or a certain amount of time, for example.

7 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 white paper accompanies a second prerelease of DAML-S, version 0.6. The prerelease materials can be found at http://www.daml.org/services/. When our work on groundings is completed, version 1.0 will be released, at the same URL.

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 out information but also get things done.

References

[1] J. F. Allen and H. A. Kautz. A model of naive temporal reasoning. In J. R. Hobbs and R. C. Moore, editors, *Formal Theories of the Commonsense World*, pages 251–268. Ablex Publishing Corp., 1985.

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

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

[4] G. Denker, J. Hobbs, D. Martin, S. Narayanan, and R. Waldinger. Accessing information and services on the daml-enabled web. In *Proc. Second Int'l Workshop Semantic Web (SemWeb'2001)*, 2001.

[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] 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.

[9] 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.

[10] 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.

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

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

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

[14] S. Narayanan. Reasoning about actions in narrative understanding. In *Proc. International Joint Conference on Artifical Intelligence (IJCAI'1999)*, pages 350–357. Morgan Kaufman Press, San Francisco, 1999.

[15] 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.

[16] K. Sycara and M. Klusch. Brokering and matchmaking for coordination of agent societies: A survey. In A. e. a. Omicini, editor, *Coordination of Internet Agents*. Springer, 2001.

[17] 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.

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