DAML-S: Semantic Markup For Web Services

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. This white paper describes 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. We also compare DAML-S with several industry efforts to define standards for characterizing services on the Web.

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 [8]. 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 white paper describes an effort 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 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 efforts in industry to standardize a markup language for services.

2 Some Motivating Tasks

Services can be 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. Or 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 of course, complex services have provided the primary motivations for the features of the language [11, 12]. The following four sample tasks will give the reader an idea of the kinds of tasks we expect DAML-S to enable.

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 [4, 19, 10], 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 appropriate Web services to perform some task, given a high-level description of the objective of the task. For example, the user may want to make all the necessary 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*

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). Ultimately, we will want to be able

[1] A service profile has also been called service capability advertisement [17]

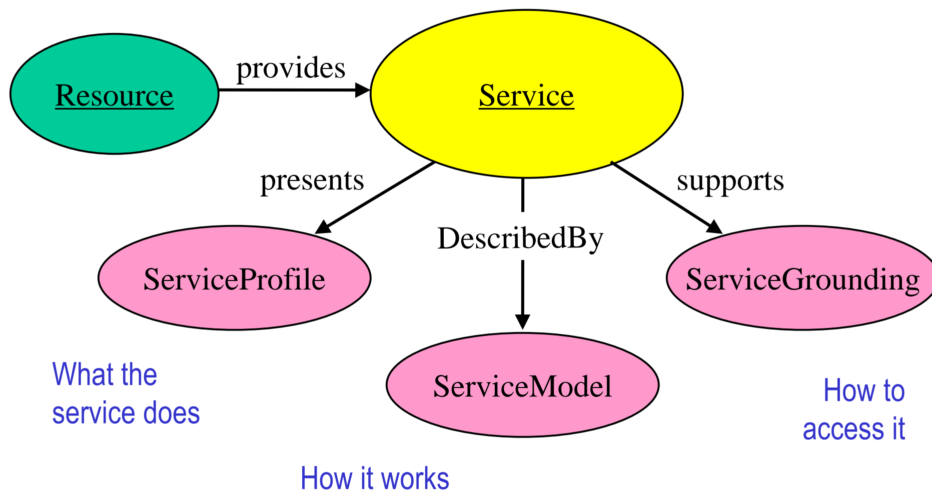

Figure 1: Top level of the service ontology

to use logical rules 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. Indeed, as DAML and DAML-S 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 [5] 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 [10]), 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. Groundings will 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 use-

ful. 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

A service profile provides a high-level description of a service and its provider [18, 17]; it is used to request or advertise services with discovery/location registries. Service profiles consist of three types of information: a human readable *description* of the service; a specification of the *functionalities* that are provided by the service; and a host of *functional attributes* which provide additional information and requirements about the service that assist when reasoning about several services with similar capabilities. Service functionalities are represented as a transformation from the inputs required by the service to the outputs 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. Functional attributes specify additional information about the service, such as what guarantees of response time or accuracy it provides, or the cost of the service.

While service providers use the service profile to advertise their services, service requesters use the profile to specify what services they need and what they expect from such a service. For instance, a requester may look for 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 against the profiles advertised by other services and identify which services provide the best match.

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 it allows them 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.

The service profile contains only the information that allows registries to decide which advertisements are matched by a request. To this extent, the information in the profile is a summary of the information in the process model and service grounding. Where, as in the above example, the service does not advertise some of its functionalities, they will not be part of the service profile. But they *are* part of the service model to the extent that they are needed for *achieving* the advertised services. For example, looking for a book is an essential prerequisite for buying it, so it would be specified in the process model, but not necessarily in the profile. Similarly, information about shipping may appear within the process model but not the profile.

4.1 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.

4.2 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="#parameter"/>
</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. Preconditions present one or more 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 events that are caused by the successful execution of a service.

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

The service profile also provides a specific type of preconditions called `accessConditions` that are expected to be true for the service to succeed, but they are not modified by the activity of the service. Access conditions are used when the access to the service is restricted to only some users: as, for example, services that are restricted to users affiliated to some organization. For instance, to access a classified news service a user needs to have some level of clearance, details about it would be specified as an `accessCondition`.

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.

4.3 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.

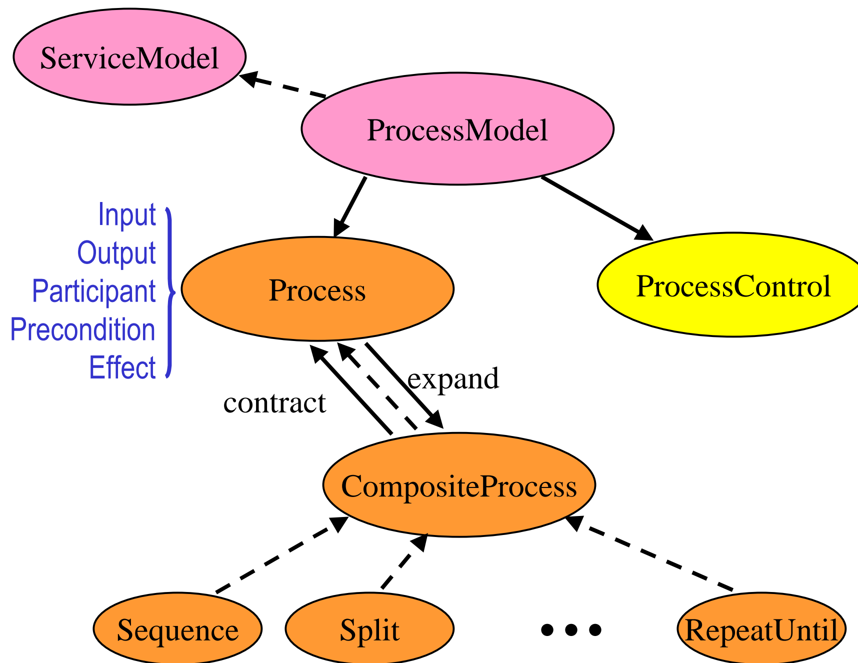

Figure 2: Top level of process modeling ontology

5 Modeling Services as Processes

A more detailed perspective on services is that a service can be viewed as a *process*. We have defined a particular subclass of *ServiceModel*, the *ProcessModel* (as shown in figure 2), 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 component actions or processes, 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. We have defined a simple ontology of time, described below; in subsequent versions this will be elaborated. We also expect in a future version to provide an ontology of resources.

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 [14, 13], emerging standards in process modeling and workflow technology such as the NIST's Process Specification Language (PSL) [16] and the Workflow Management Coalition effort (http://www.aiim.org/wfmc), work on modeling verb semantics and event structure [15], work in AI on modeling complex actions [9], and work in agent communication languages [10, 7].

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

[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".

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. Participants and other parameters may be specified; for example, the participants may include the roles in the event frame, such as the agents, patient, and instrument, whereas other parameters, especially for physical devices, might be 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.

A process can often be viewed either as a primitive, undecomposable process or as a composite process, decomposable into a complex of other primitive or composite processes. Either perspective may be the more useful in some given context. Thus, a top-level *Process* class has, as its sole subclass, *CompositeProcess*, which in turn is subclassed by a variety of control structures.

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*, *output*, *participant*, *precondition*, and (conditional) *effect*. Each of these properties ranges over a DAML object, which, at the upper ontology level, is not restricted at all. The properties *input*, *output*, and *participant* are categorized as subproperties of *parameter*. 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 is an example of a property definition:

  ```
  <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 its 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.

- **CompositeProcess**

  ```
  <rdfs:Class rdf:ID="CompositeProcess">
    <rdfs:subClassOf rdf:resource="#Process"/>
  </rdfs:Class>

  <rdf:Property rdf:ID="components">
    <rdfs:comment>
  ```

Ultimately, however, the choice is arbitrary. It is modeled after computational procedures or planning operators.

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

Composite processes are processes that have additional properties called *components* to indicate the ordering and conditional execution of the subprocesses from which they are composed. For instance, the composite process, *Sequence*, has a *components* property that ranges over a *ProcessList* (a list whose items are restricted to be simple or composite processes). In the process "upper ontology", we have attempted to come up with a minimal set of process classes 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.

Note that while a composite process is a process, and thus has slots for preconditions and effects, there may be no easy way to compute these values for an arbitrary composite process, given its component sub-processes.

There are two fundamental relations between processes and composite processes. The *expand* relation associates a Process with the CompositeProcess describing its component subprocesses, while its inverse, the *collapse* relation represents the association of the CompositeProcess to its atomic Process form. Expanding is intended to provide a "glassbox" and collapsing a "blackbox" view of the process. The expanded version is likely to be used for service composition (both off-line and runtime) and the collapsed version for service execution.

The minimal set of composition templates (subclasses of *CompositeProcess*) is as follows:

Sequence : A list of Processes to be done in order. We use a DAML restriction to restrict the components of a Sequence process to be a List of subprocesses (simple and/or composite).

```
<rdfs:Class rdf:ID="Sequence">
 <rdfs:subClassOf rdf:resource="#CompositeProcess"/>
  <rdfs:subClassOf>
   <daml:Restriction>
    <daml:onProperty rdf:resource="#components"/>
    <daml:toClass rdf:resource="#ProcessList"/>
  </daml:Restriction>
 </rdfs:subClassOf>
</rdfs:Class>
```

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

```
<rdfs:Class rdf:ID="Split">
 <rdfs:subClassOf rdf:resource="#CompositeProcess"/>
  <rdfs:subClassOf>
   <daml:Restriction>
    <daml:onProperty rdf:resource="#components"/>
    <daml:toClass rdf:resource="#ProcessBag"/>
  </daml:Restriction>
 </rdfs:subClassOf>
</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 processes can be executed in any order. No further constraints are specified. All processes must be executed.

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

Choice : Choice is a composite process with additional properties "chosen" and "chooseFrom". These properties can be used both for process and execution control (for example, 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.

Condition : Conditions are composite processes with an output property (conditionValue) whose range is a binary value. Conditions usually correspond to test actions, but they may be world states, resource levels, timeouts or other things affecting the evolution of processes.

If-Then-Else : The If-Then-Else Class is a composite process that has properties condition, then and else holding different aspects of the If-Then-Else composite process. Its semantics is intended as "Test If-condition; if True do *Then*, if False do *Else*."

```
<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="#CompositeProcess"/>
</rdf:Property>

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

Iterate : Iterate is a composite process whose next process property has the same value as the current process. 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.[4]

Repeat-Until : The Repeat-Until class is similar to the Repeat-While class in that specializes the If-Then-Else class where the "ifCondition" is the same as the untilCondition and different

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

[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.

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.

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. 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 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. We have released an initial version of DAML-S. It can be found at the URL: http://www.daml.org/services/daml-s

We expect to enhance it in the future in ways that we have indicated in the 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] M. Ciocoiu, G. M., and D. Nau. Ontologies for integrating engineering applications. To appear in *Journal of Computing and Information Science in Engineering*, 2001.

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

[5] 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.

[6] M. G. 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] T. Finin, Y. Labrou, and J. Mayfield. KQML as an agent communication language. In J. Bradshaw, editor, *Software Agents*. MIT Press, Cambridge, 1997.

[8] J. Hendler and D. L. McGuinness. Darpa agent markup language. *IEEE Intelligent Systems*, 15(6):72–73, 2001.

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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