DAML-S: Semantic Markup For Web Services

The DAML Services Coalition[1]:
Anupriya Ankolekar[2], Mark Burstein[3], Jerry R. Hobbs[4], Ora Lassila[5],
David L. Martin[4], Sheila A. McIlraith[6], Srini Narayanan[4], Massimo Paolucci[2],
Terry Payne[2], Katia Sycara[2], Honglei Zeng[6]

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 DAML-S, a DAML+OIL ontology for describing the properties and capabilities of web services, and that supports the automatic discovery, invocation, composition and monitoring of these services. In this 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. 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] through the development of Semantic Web markup languages such as DAML+OIL [10]. Such markup languages enable the creation of arbitrary domain ontologies that support unambiguous description of web content. Web services are resources 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. Software agents should be able to locate, select, employ, compose, and monitor Web-based services automatically. To achieve this, an agent needs a computer-interpretable description of the service and how it can be invoked.

Web-based services are garnering a great deal of interest from industry, and standards are being developed for low-level descriptions of these services. Languages such as WSDL (Web Services Description Language) provide a communication level description of messages and simple protocols used by services. An important goal for DAML is to complement this effort by establishing a higher-level framework within which semantic descriptions of services can be 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.

[1]Authors' names are in alphabetical order.
[2]The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA
[3]BBN Technologies, Cambridge, Massachusetts
[4]Artificial Intelligence Center, SRI International, Menlo Park, California
[5]Nokia Research Center, Burlington, Massachusetts
[6]Knowledge Systems Laboratory, Stanford University, Stanford, California

This paper describes a collaborative effort by BBN Technologies, Carnegie Mellon University, Nokia, Stanford University, and SRI International 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 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. For example, one's interaction with a book retail service such as *Amazon.com* might proceed as follows: 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 illustrate the kinds of tasks we expect DAML-S to enable [13, 14].

1. **Automatic Web Service Discovery.** This 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 a *Middle Agent* [5, 12, 24], so that requesters can find this service by querying 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.** This 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?* This is answered by the "profile"[7]. Thus, the class *Service* `presents` a *ServiceProfile*.

[7] A service profile has also been called service capability advertisement [20].

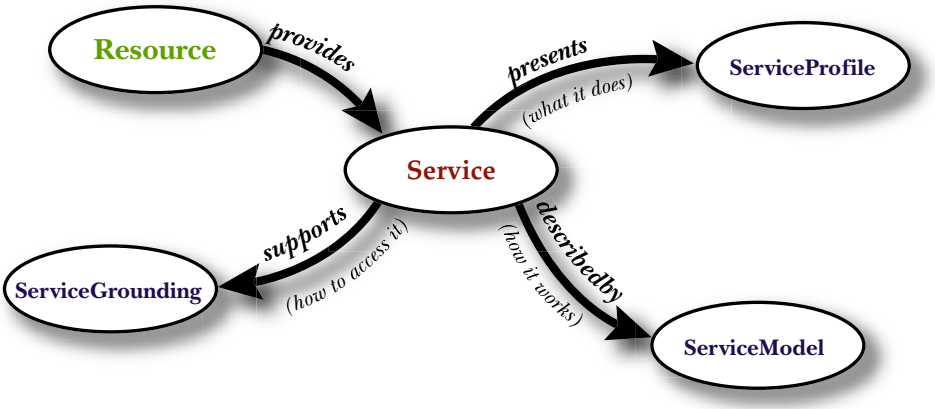

Figure 1: Top level of the service ontology

- *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). 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+OIL 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 [6] 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 [12]), 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 (i.e., the marshaling/serialization techniques employed).

Generally speaking, the *ServiceProfile* provides the information needed for an agent to discover a service. Taken together, the *ServiceModel* and *ServiceGrounding* objects associated with a service provide enough information for an agent to make use of a service. The upper ontology for services 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. Furthermore, it should be possible for a service to offer multiple profiles, models, and/or groundings. Hence, no maximum cardinality is specified. In addition, 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.

The service profile and the service model are described in detail in the following two sections. A discussion of the grounding and it's synergy with WSDL can be found in [4].

4 Service Profiles

A service profile provides a high-level description of a service and its provider [21, 20]; it is used to request or advertise services with discovery services and capability 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 *functional attributes* which provide additional information that may assist when reasoning about 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 define advertisements for their services using the profile ontology, service requesters utilize the profile to specify what services they need and what they expect from such a service. For instance, a requester may look for a service that reports current market and stock quotes. Middle Agents and other discovery services then match the request against the advertised profiles, 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.

The following subsections break down the type of information represented by the profile into the three categories: *description*, *functionalities* and *functional attributes*.

4.1 Description

The profile includes high-level information about the service and it's provenance. This information has two roles: to describe the service and the service provider (or requester) in a human readable format; and to provide semi-structured semantics that may support service matchmaking. Typically, this information would be presented to users when browsing a service registry. The *description* properties are listed in Table 1.

Table 1: Description Properties - defining the provenance of the service.

Property	**Description**
`serviceName`	The name of the service.
`intendedPurpose`	A high-level description on what constitutes successful accomplishment of a service execution.
`textDescription`	A brief, human readable description of the service, summarizing what the service offers or what capabilities are being requested.
`role`	An abstract link to *Actors* that may be involved in the service execution.
`requestedBy`	A sub-property of `role` that links the service to the *Actor* that requests a service.
`providedBy`	A sub-property of `role` that links the service to the *Actor* that advertised the service.

The class *Actor* is also defined to describe entities (e.g. humans or organizations) that provide or request web services. Two specific classes are derived from the *Actor* class; the *ServiceRequester* class and *ServiceProvider* class, to represent the requester and provider of the service respectively. Properties of *Actor* include `physicalAddress`, `webURL`, `name`, `phone`, `email`, and `fax`.

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 addi-

tion, 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 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.

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. While 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 Web site in general, one cannot reasonably expect to do this from California. The functional attributes described in Table 2 address the problem that there are properties that can be used to describe a service other than as a functional process.

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

Table 2: Functional Attributes

Property	Description
`geographicRadius`	The geographic scope of the service, either at the global scale (e.g. for e-commerce) or at a regional scale (e.g. pizza delivery).
`degreeOfQuality`	Quality qualifications, such as providing the `cheapest` or `fastest` possible service.
`serviceParameter`	An expandable list of properties that characterize the execution of a service, such as `averageResponseTime` or `invocationCost`.
`communicationThru`	High-level summary of how a service may communicate, e.g. what agent communication language (ACL) is used (e.g., FIPA, KQML, SOAP). This summarizes, but does not replace, the descriptions provided by the service grounding.
`serviceType`	Broad classification of the service that might be described by an ontology of service types, such as B2B, B2C etc.
`serviceCategory`	Categories defined within some service category ontology. Such categories may include *Products*, *Problem-Solving*, *Commercial Services* etc.
`qualityGuarantees`	Guarantees that the service promises to deliver, e.g. guaranteeing to provide the lowest possible interest rate, or a response within 3 minutes, etc.
`qualityRating`	Industry-based ratings, such as the "Dun and Bradstreet Rating" for businesses, or the "Star Rating" for Hotels.

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 model*, 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 [9], work in programming languages and distributed systems [16, 15], emerging standards in process modeling and workflow technology such as the NIST's Process Specification Language (PSL) [19] and the Workflow Management Coalition effort (http://www.aiim.org/wfmc), work on modeling verb semantics and event structure [17], previous work on action-inspired Web service markup [14], work in AI on modeling complex actions [11], and work in agent communication languages [12, 8].

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

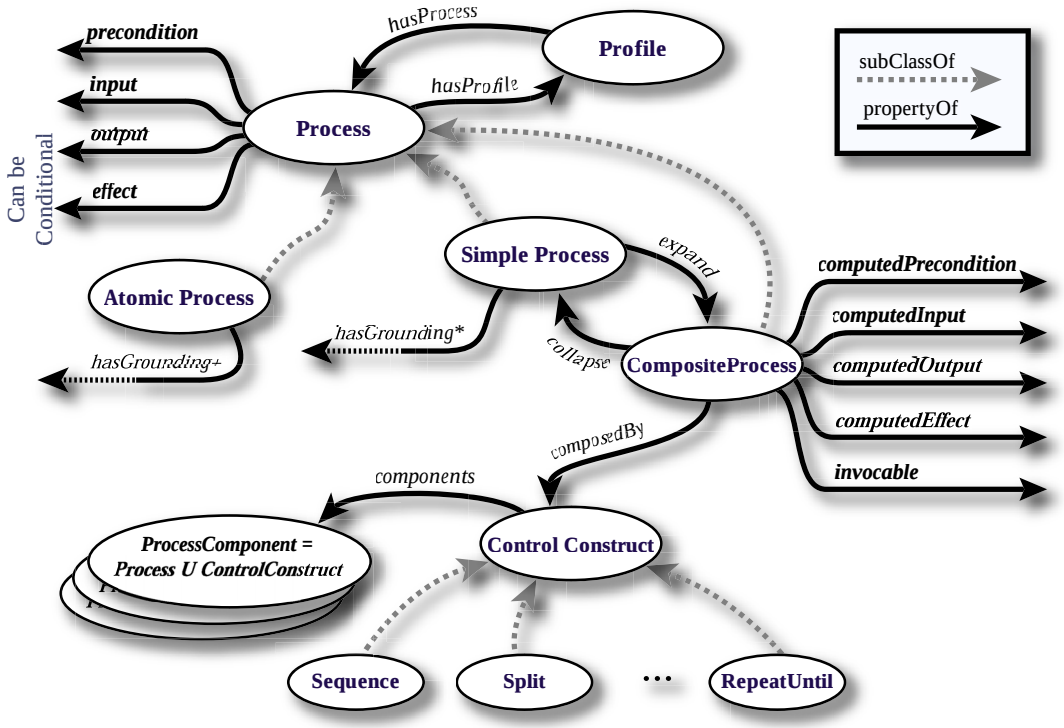

Figure 2: Top level of process modeling ontology

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, we have:

- **Process**

  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.

```
<rdfs:Class rdf:ID="Process">
  <rdfs:comment>
    Top-level class for describing how a service works
  </rdfs:comment>
</rdfs:Class>
```

  In addition to its action-related properties, a *Process* has a number of bookkeeping properties such as `name`, `address`, `documentsRead`, `documentsUpdated`, and so on.

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

- **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>
```

- **SimpleProcess**

  Simple processes, on the other hand, are not normally invocable and not normally 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"/>
```

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 *ProcessComponent*s, 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*. In the following, due to space constraints, we give (partial) code samples only for *Sequence* and *If-Then-Else*.

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. The components are restricted similarly to those of *Sequence*, except to a bag rather than a list.

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" [8], 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 `ifCondition`; 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.) We show here just the property definitions.

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

<rdf:Property rdf:ID="then">
```

[8] 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)($0 \leq n \leq |components(chooseFrom)|, 0 < m \leq |components(chosen)|$).

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

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

Iterate : *Iterate* is a control construct whose `nextProcessComponent` property has the same value as the current process component. *Repeat* is defined as a synonym of the *Iterate* class. The repeat/iterate process makes no assumption about how many iterations are made or when to initiate, terminate or resume. The initiation, termination or maintenance condition could be specified with a `whileCondition` or an `untilCondition` as below.[9]

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 *TemporalEntity*.

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

[9] 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 `startOf` property whose domain is the *Interval* class and whose range is an *Instant*.

2. The `endOf` 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 occurring 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 `atTime` 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 `startOf` and `endOf` 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 *TemporalEntity* class and whose range is a *TemporalEntity*.

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 Example Walk-Through

To illustrate the concepts described in this paper, we have developed an example of a fictitious book-buying service offered by the Web service provider, Congo Inc. Congo has a suite of programs that they are making accessible on the Web. Congo wishes to compose these individual programs into Web services that it offers to its users. We focus here on the Web

service of buying a book, CongoBuy. In the DAML-S release, we present a walk-through that steps through the process of creating DAML-S markup for Congo[10].

We take the perspective of the typical Web service provider and consider three automation tasks that a Web service provider might wish to enable with DAML-S: 1) automatic Web service discovery, 2) automatic Web service invocation, and 3) automatic Web service composition and interoperation. For the purposes of this paper, we limit our discussion to the second and third tasks.

6.1 Web Service Invocation

To automate Web Service Invocation, DAML-S markup must tell a program how to automatically construct an (http) call to execute or invoke a Web service, and what output(s) may be returned from the service. To enable such functionality, the process ontology in DAML-S provides markup to describe individual and composite Web-accessible programs as either simple or composite processes.

6.1.1 Define the Service as a Process

Congo Inc. provides the CongoBuy Web service to its customers. We view the CongoBuy Web service as a simple process, i.e., it is a subclass of the class *SimpleProcess* in the process ontology.

```
<rdfs:Class rdf:ID="CongoBuy">
  <rdfs:subClassOf rdf:resource=
    "http://www.daml.org/services/.../Process.daml#SimpleProcess"/>
</rdfs:Class>
```

Although the CongoBuy service is actually a predetermined composition of several of Congo's Web-accessible programs, it is useful to initially view it as a black-box process. The black-box process, CongoBuy has a variety of invocation-relevant properties, including input, (conditional) output and parameter. For example, input to the CongoBuy book-buying service includes the name of the book (bookName), the customer's credit card number, and their account number and password. If the service being described is simple in that it is not the composition of other services or programs, then the service inputs are simply the set of inputs that must be provided in the service invocation. The outputs are the outputs returned from the service invocation. Note that these outputs may be conditional. For example the output of a book-buying service will vary depending upon whether the book is in or out of stock.

In contrast, if the service is composed of other services, as is the case with CongoBuy, then the rationale for specification of the inputs, outputs and parameters is more difficult, and the utility of these properties is limited. In the simplest case, the inputs and outputs of the black-box process can be defined to be the composition of all the possible inputs and all the possible (conditional) outputs of the simple services that the black-box process may invoke, taking every possible path through the composition of simple services. Note however that this is not a very exacting specification. In particular, the collection of outputs may be contradictory (e.g., one path of CongoBuy may lead to confirmation of a purchase, while another may lead to confirmation of no purchase). The conditions under which inputs and

[10]This Congo example can be found at http://www.daml.org/services/daml-s/2001/06/Congo.daml

outputs arise are encoded exactly in the expand of this black-box process, and can be retrieved from the expanded process. The inputs, outputs and parameters for the black-box process are designed to be a useful shorthand. Thus, it could be argued that the inputs and outputs should describe the most likely inputs and outputs through the system. However, in some cases, even this is difficult to define. For now, DAML-S leaves this decision up to the Web service provider.

The following is an example of one input to CongoBuy. Note that it is a subproperty of the property `input` of *Process*.

```
<rdf:Property rdf:ID="bookName">
  <rdfs:subPropertyOf rdf:resource=
    "http://www.daml.org/services/daml-s/2001/06/Process.daml#input"/>
  <rdfs:domain rdf:resource="#CongoBuy"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/10/XMLSchema#string"/>
</rdf:Property>
```

An output can similarly be defined as a subproperty of the property `output` of *Process*. In a real book-buying service, this output would likely be conditioned on the book being in stock, or the customer's credit card being valid, but to simplify our example, we assume Congo has an infinite supply of books, and infinite generosity.

```
<rdf:Property rdf:ID="eReceiptOutput">
  <rdfs:subPropertyOf rdf:resource=
    "http://www.daml.org/services/daml-s/2001/06/Process.daml#output"/>
  <rdfs:range rdf:resource="#EReceipt"/>
</rdf:Property>
```

In addition to input and output properties, each service has parameter properties. A parameter is something that affects the outcome of the process, but which is not an input provided by the invoker of the process. It may be known by the service, or retrieved by the service from elsewhere. For example, the fact that the customer's credit card is valid, is a parameter in our CongoBuy process, and is relevant when considering the use of the CongoBuy, but it is not an input or output of CongoBuy.

```
<rdf:Property rdf:ID="creditCardValidity">
  <rdfs:subPropertyOf rdf:resource=
    "http://www.daml.org/services/.../Process.daml#parameter"/>
  <rdfs:range rdf:resource="#ValidityType"/>
</rdf:Property>
```

6.1.2 Define the Process as a Composition of Processes

Given the variability in the specification of inputs, outputs and parameters, it is generally insufficient to simply specify a service as a black-box process, if the objective is to automate service invocation. We must expand the black-box service to describe its composite processes. This is achieved by first defining the individual processes and then defining their composition as a composite process.

Define the Individual Processes

We define each of the simple services in CongoBuy, i.e., LocateBook, PutInCart, etc.[11]

```
<rdfs:Class rdf:ID="LocateBook">
  <rdfs:subClassOf rdf:resource=
    "http://www.daml.org/services/.../Process.daml#SimpleProcess"/>
</rdfs:Class>

<rdfs:Class rdf:ID="PutInCart">
  <rdfs:subClassOf rdf:resource=
    "http://www.daml.org/services/.../Process.daml#SimpleProcess"/>
</rdfs:Class>

<rdf:Property rdf:ID="bookSelected">
  <rdfs:subPropertyOf rdf:resource=
    "http://www.daml.org/services/.../Process.daml#input"/>
  <rdfs:domain rdf:resource="#PutInCart"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/10/XMLSchema#string"/>
</rdf:Property>
```

Define the Composition of the Individual Processes

The composition our simple services can be defined by using the composition constructs created in the process ontology, i.e., *Sequence, Split, Split + Join, Unordered, Condition, If-Then-Else, Repeat-While, Repeat-Until*. Using these constructs, we specify a *CompositeProcess*, ExpandedCongoBuy, which is composed from simple services such as those shown above. For lack of space, the definition of ExpandedCongoBuy is omitted here, but may be seen at the URL mentioned in the last footnote.

Having defined ExpandedCongoBuy, its relationship to CongoBuy can be indicated using the `expand` (and `collapse`) properties.

6.1.3 Automated Service Composition and Interoperation

The DAML-S markup required to automate service composition and interoperation builds directly on the markup for service invocation. In order to automate service composition and interoperation, we must also encode the effects a service has upon the world, and the preconditions for performing that service. For example, when a human being goes to www.congo.com and successfully executes the CongoBuy service, the human knows that they have purchased a book, that their credit card will be debited, and that they will receive a book at the address they provided. Such consequences of Web service execution are not part of the input/output markup we created for automating service invocation.

The process ontology provides precondition and effect properties of a process to encode this information. As with our markup for automated service invocation, we define preconditions and effects both for the black-box process CongoBuy and for each of the simple processes that define its composition, and as with defining inputs and outputs, it is easiest to define the preconditions and effects for each of the simple processes first, and then to aggregate them into preconditions and effects for CongoBuy. The markup is analogous to the markup for input and (conditional) output, but is with respect to the properties precondition and (conditional) effect, instead.

[11] Additional DAML code is needed here to specify the relationship between the bookName property of CongoBuy and the bookSelected property of PutInCart. As of this writing, discussions are underway to determine the best way to indicate this relationship in DAML+OIL.

7 Related Efforts

Industry efforts to develop standards for electronic commerce, and in particular for the description of Web-based services currently revolve around UDDI, WSDL, and ebXML [23]. There have also been company-specific initiatives to define architectures for e-commerce, most notably E-speak from Hewlett-Packard.

Nevertheless, we believe that DAML-S provides functionality that the other efforts do not. In comparison to the DAML-S characterization of services, the industry standards mostly focus on presenting a *ServiceProfile* and a *ServiceGrounding* of services (to use DAML-S terminology). *ServiceGroundings* are supported by all the standards. However, they are limited with respect to DAML-S profiles in that they cannot express logical statements, e.g. preconditions and postconditions, or rules to describe dependencies between the profile elements. Input and output types are supported to varying extents. Furthermore, DAML-S supports the description of certain functional attributes of services, which are not covered in the other standards, such as qualityGuarantees and serviceType.

With respect to the four tasks of automatic Web service discovery, automatic Web service invocation, automatic Web service interoperation and composition, and automatic Web service execution monitoring that DAML-S is meant to support, the standards primarily enable the first and the second tasks to a certain extent. These standards are still evolving and it is unclear at present to what extent composition will be addressed. At the moment, the standards do not consider the *ServiceModel* of a service and thus, they also do not support execution monitoring, as defined in this paper.

In the following sections, we look in greater detail at each of these technologies in turn and compare them to DAML-S.

7.1 UDDI

UDDI (Universal Description, Discovery and Integration) is an initiative proposed by Microsoft, IBM and Ariba to develop a standard for an online registry, and to enable the publishing and dynamic discovery of Web services offered by businesses [22]. UDDI allows programmers and other representatives of a business to locate potential business partners and form business relationships on the basis of the services they provide. It thus facilitates the creation of new business relationships.

The primary target of UDDI seems to be integration and at least semi-automation of business transactions in B2B e-commerce applications. It provides a registry for registering businesses and the services they offer. These are described according to an XML schema defined by the UDDI specification. A Web service provider registers its advertisements along with keywords for categorization. A Web services user retrieves advertisements out of the registry based on keyword search. The UDDI search mechanism relies on pre-defined categorization through keywords and does not refer to the semantic content of the advertisements. The registry is supposed to function in a fashion similar to white pages or yellow pages, where businesses can be looked up by name or by a standard service taxonomy as is already used within the industry. UDDI attempts to cover all kinds of services offered by businesses, including those that are offered by phone or e-mail and similar means; in principle, DAML-S could do this, but it has not been our focus.

Technically speaking, each business description in UDDI consists of a businessEntity element, akin to a White Pages element describing the contact information for a business. A busi-

nessEntity describes a business by name, a key value, categorization, services offered (businessService elements) and contact information for the business. A businessService element describes a service using a name, key value, categorization and multiple "bindingTemplate" elements. This can be considered to be analogous to a Yellow Pages element that categorises a business. A bindingTemplate element in turn describes the kind of access the service requires (phone, mailto, http, ftp, fax etc.), key values and tModelInstances. tModelInstances are used to describe the protocols, interchange formats that the service comprehends, that is, the technical information required to access the service. It is also used to describe the "namespaces" for the classifications used in categorization. Many of the elements are optional, including most of the ones that would be required for matchmaking or service composition purposes.

UDDI aims to facilitate the discovery of potential business partners and the discovery of services and their groundings that are offered by known business partners. This may or may not be done automatically. When this discovery occurs, programmers affiliated with the business partners program their own systems to interact with the services discovered. This is also the model generally followed by ebXML. DAML-S enables more flexible discovery by allowing searches to take place on almost any attribute of the ServiceProfile. UDDI, in contrast, allows technical searches only on tModelKeys, references to tModelInstances, which represent full specifications of a kind of service.

UDDI does not support semantic descriptions of services. Thus, depending on the functionality offered by the content language, although agents can search the UDDI registry and retrieve service descriptions, a human needs to be involved in the loop to make sense of the descriptions, and to program the access interface.

Currently, UDDI does not provide or specify content languages for advertisement. Although WSDL is most closely associated with UDDI as a content language, the specification refers to ebXML and XML/edi also as potential candidates. Content languages could be a possible bridge between UDDI and DAML-S. DAML-S is also a suitable candidate for a content language and in this sense, DAML-S and UDDI are complementary. A higher-level service or standard defined on top of UDDI could take advantage of the additional richness of content DAML-S has to offer within the UDDI registries.

7.2 WSDL

WSDL (Web Services Description Language) is an XML format, closely associated with UDDI as the language for describing interfaces to business services registered with a UDDI database. Thus, it is closer to DAML-S in terms of functionality than UDDI. Like DAML-S, it attempts to separate services, defined in abstract terms, from the concrete data formats and protocols used for implementation, and defines bindings between the abstract description and its specific realization [3]. However, the abstraction of services is at a lower level than in DAML-S.

Services are defined as sets of ports, i.e. network addresses associated with certain protocols and data format specifications. The abstract nature of a service arises from the abstract nature of the messages and operations mapped to a port and define its port type. Port types are reusable and can be bound to multiple ports [18]. There are four basic types of operations in WSDL: a one-way, a (two-way) request-response, a (two-way) solicit-response and a (one-way) notification message. A message itself is defined abstractly as a request, a response or even a parameter of a request or response and its type, as defined in a type system like XSD.

They can be broken into parts to define the logical break-down of a message.

Messages and operations are defined abstractly and are thus reusable and extensible and correspond roughly to the DAML-S ServiceProfile. The service element itself incorporates both a ServiceProfile and ServiceGrounding information. WSDL service descriptions are not as expressive as DAML-S profiles. Preconditions, postconditions and effects of service access cannot be expressed within WSDL.

Like UDDI, WSDL does not support semantic description of services. WSDL focuses on the grounding of services and although it has a concept of input and output types as defined by XSD, it does not support the definition of logical constraints between its input and output parameters. Thus its support for discovery and invocation of services is less versatile than that of DAML-S.

7.3 E-speak

Hewlett-Packard is collaborating with the UDDI consortium to bring E-speak technology to the UDDI standard. E-speak and UDDI have similar goals in that they both facilitate the advertisement and discovery of services. E-speak is also comparable to WSDL in that it supports the description of service and data types [7]. It has a matching service that compares service requests with service descriptions, primarily on the basis of input-output and service type matching.

E-speak describes services (known as “Resources”) as a set of attributes within several “Vocabularies”. Vocabularies are sets of attributes common to a logical group of services. E-speak matches lookup requests against service descriptions with respect to these attributes. Attributes take common value types such as String, Int, Boolean and Double. There is a base vocabulary which defines basic attributes such as Name, Type (of value String only), Description, Keywords and Version. Currently, there is no semantic meaning attached to any of the attributes. Any matching which takes place is done over the service description attributes which does not distinguish between any further subtypes. DAML-S had a much richer set of attributes; in DAML-S terminology, the input/output parameters, effects and additional functional attributes. In addition, dependencies between attributes and logical constraints on them are not expressible within E-speak.

Unlike UDDI, which was intended to be an open standard from the beginning, e-speak scores relatively low on interoperability. It requires that an e-speak engine be run on all participating client machines. Furthermore, although e-speak is designed to be a full platform for Web services and could potentially expose a execution monitoring interface, service processes remain a black-box for the e-speak platform and consequently no execution monitoring can be done.

7.4 ebXML

ebXML, being developed primarily by OASIS and the United Nations, approaches the problem from a workflow perspective. ebXML uses two views to describe business interactions, a Business Operational View (BOV) and a Functional Service View (FSV) [23]. The BOV deals with the semantics of business data transactions, which include operational conventions, agreements, mutual obligations and the like between businesses. The FSV deals with the supporting services: their capabilities, interfaces and protocols. Although ebXML does

not concentrate on only Web services, the focus of this view is essentially the same as that of the current DAML-S effort.

It has the concept of a Collaboration Protocol Profile (CPP) "which allows a Trading Partner to express their supported Business Processes and Business Service Interface requirements [such that they are understood] by other ebXML compliant Trading Partners", in effect a specification of the services offered by the Trading Partner. A Business Process is a set of business document exchanges between the Trading Partners. CPPs contain industry classification, contact information, supported Business Processes, interface requirements etc. They are registered within an ebXML registry, in which there is discovery of other Trading Partners and the Business Processes they support. In this respect, UDDI has some similarities with ebXML. However, ebXML's scope does not extend to the manner in which the business documents are specified. This is left to the Trading Partners to agree upon a priori by the creation of a Collaboration Protocol Agreement.

In conclusion, the kind of functionality, interoperability and dynamic matchmaking capabilities provided by DAML-S is only partially supported, as the standards are currently positioned, by WSDL and UDDI. UDDI may become more sophisticated as it incorporates e-speak-like functionalities, but it will not allow automatic service interoperability until it incorporates the information provided by DAML-S.

8 Summary and Current Status

DAML-S is an attempt to provide an ontology, within the framework of the DARPA Agent Markup Language, for describing Web services. It will enable users and software agents to automatically discover, invoke, compose, and monitor Web resources offering services, under specified constraints. We have released an initial version of DAML-S. It can be found at the URL: http://www.daml.org/services

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.

Acknowledgments

The authors have profited from discussions about this work with Ron Fadel, Richard Fikes, Jessica Jenkins, James Hendler, Mark Neighbors, Tran Cao Son, and Richard Waldinger. The research was funded by the Defense Advanced Research Projects Agency as part of the DARPA Agent Markup Language (DAML) program under Air Force Research Laboratory contract F30602-00-C-0168 to SRI International, F30602-00-2-0579-P00001 to Stanford University, and F30601-00-2-0592 to Carnegie Mellon University. Additional funding was provided by Nokia Research Center.

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

[4] DAML-S Coalition:, A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: Web Service Description for the Semantic Web. Submitted to the First International Semantic Web Conference (ISWC01), 2002.

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

[6] 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.

[7] E-Speak and the HP Web Services Platform. http://www.e-speak.net/, 2001.

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

[9] 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.

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

[11] 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.

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

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

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

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

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

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

[18] U. Ogbuji. Using WSDL in SOAP applications: An introduction to WSDL for SOAP programmers. http://www-106.ibm.com/developerworks/library/ws-soap/?dwzone=ws, 2001.

[19] 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.

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

[21] 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.

[22] UDDI. The UDDI Technical White Paper. http://www.uddi.org/, 2000.

[23] D. Webber and A. Dutton. Understanding ebXML, UDDI and XML/edi. http://www.xml.org/feature_articles/2000_1107_miller.shtml, 2000.

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