DAML-S: Web Service Description for the Semantic Web

DAML-S Coalition: Anupriya Ankolekar[2], Mark Burstein[1], Jerry R. Hobbs[4], Ora Lassila[3], David Martin[4], Drew McDermott[6], Sheila A. McIlraith[5], Srini Narayanan[4], Massimo Paolucci[2], Terry Payne[2], Katia Sycara[2]

[1] BBN Technologies
[2] Carnegie Mellon University
[3] Nokia Research Center
[4] SRI International
[5] Stanford University
[6] Yale University

Abstract. In this paper we present DAML-S, a DAML+OIL ontology for describing the properties and capabilities of Web Services. Web Services – Web-accessible programs and devices – are garnering a great deal of interest from industry, and standards are emerging for low-level descriptions of Web Services. DAML-S complements this effort by providing Web Service descriptions at the application layer, describing *what* a service can do, and not just *how* it does it. In this paper we describe three aspects of our ontology: the service profile, the process model, and the service grounding. The paper focuses on the grounding, which connects our ontology with low-level XML-based descriptions of Web Services.

1 Services on the Semantic Web

The Semantic Web [2] is rapidly becoming a reality through the development of Semantic Web markup languages such as DAML+OIL [9]. These markup languages enable the creation of arbitrary domain ontologies that support the unambiguous description of Web content. Web Services [15] – Web-accessible programs and devices – are among the most important resources on the Web, not only to provide information to a user, but to enable a user to effect change in the world. Web Services are garnering a great deal of interest from industry, and standards are being developed for low-level descriptions of Web Services. Languages such as WSDL (Web Service Description Language) provide a communication level description of the messages and protocols used by a Web Service. To complement this effort, our interest is in developing semantic markup that will sit at the application level above WSDL, and describe *what* is being sent across the wires and *why*, not just *how* it is being sent.

We are developing a DAML+OIL ontology for Web Services, called DAML-S [5], with the objective of making Web Services computer-interpretable and hence enabling the following tasks [15]: **discovery**, i.e. locating Web Services (typically through a registry service) that provide a particular service and that adhere to specified constraints; **invocation** or activation and execution of an identified service by an agent or other service; **interoperation**, i.e. breaking down interoperability barriers through semantics, and the automatic insertion of *message parameter translations* between clients and services [10, 13, 22]; **composition** of new services through automatic selection, composition and interoperation of existing services [15, 14]; **verification** of service properties [19]; and **execution monitoring**, i.e. tracking the execution of complex or composite tasks performed by a service or a set of services, thus identifying failure cases, or providing explanations of different execution traces. 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. This paper describes a collaborative effort by BBN Technologies, Carnegie Mellon University, Nokia, Stanford University, SRI International, and Yale University, to define the DAML-S Web Services ontology. An earlier version of the DAML-S specification is described in [5]; an updated version of DAML-S is presented at `http://www.daml.org/services/daml-s/2001/10/`. In this paper we briefly summarize and update this specification, and discuss the important problem of the *grounding*, i.e. how to translate what is being sent in a message to or from a service into how it is to be sent. In particular, we present the linking of DAML-S to the Web Services Description Language (WSDL). DAML-S complements WSDL, by providing an abstract or application level description lacking in WSDL.

2 An Upper Ontology for Services

In DAML+OIL, abstract categories of entities, events, etc. are defined in terms of *classes* and *properties*. DAML-S defines a set of classes and properties, specific to the description of services, within DAML+OIL. The class SERVICE is at the top of the DAML-S ontology. Service properties at this level are very general. 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.

The ontology of services provides two essential types of knowledge about a service, characterized by the questions:

– *What does the service require of agents, and provide for them?* This is provided by the *profile*, a class that describes the capabilities and parameters of the service. We say that the class SERVICE *presents* a SERVICEPROFILE.
– *How does it work?* The answer to this question is given in the *model*, a class that describes the workflow and possible execution paths of the service. Thus, the class SERVICE is *describedBy* a SERVICEMODEL

The SERVICEPROFILE provides information about a service that can be used by an agent to determine if the service meets its rough needs, and if it satisfies constraints such as security, locality, affordability, quality-requirements, etc. In contrast, the SERVICEMODEL enables an agent to: (1) perform a more in-depth analysis of whether the service meets its needs; (2) compose service descriptions from multiple services to perform a specific task; (3) coordinate the activities of different agents; and (4) monitor the execution of the service. Generally speaking, the SERVICEPROFILE provides the information needed for an agent to discover a service, whereas the SERVICEMODEL provides enough information for an agent to make use of a service. In the following sections we discuss the service profile and the service model in greater detail, and introduce the *service grounding*, which describes how agents can communicate with and thus invoke the service.

3 Service Profile

A service profile provides a high-level description of a service and its provider [23, 24]; it is used to request or advertise services with discovery services and capability registries. Service profiles consist of three types of information: a *description* of the service and the service provider; the *functional behavior* of the service; and several *functional attributes* tailored for automated service selection.

The profile includes a high-level *description* about the service and it's provenance, which typically would be presented to users when browsing a service registry (see Table 1). 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 *Service-Requester* class and *Service-Provider* class, to represent the requester and provider of the service respectively. Properties of *Actor* include `physicalAddress`, `WebURL`, `name`, `phone`, `email`, and `fax`. *Functional attributes* specify additional information about the service, such as what guarantees of response time or accuracy it provides, the cost of the service, or the classification of the service in some registry such as the NAICS [3].

Implicitly, service profiles specify the intended purpose of the service, because they specify only those *functional behaviors* that are publicly provided. A book-selling service may involve two different functionalities: it allows clients to browse its site to find books of interest, and it allows them to buy the books they find. The book-seller has the choice of advertising just the book-buying service or may also advertise browsing functionality. In the latter case the service publicizes the fact that agents may browse without buying a book. In contrast, by advertising only the book-selling functionality, the service discourages browsing by requesting agents that do not intend to buy.

While service providers define advertisements for their services using the service profile, service requesters also use the profile to specify their needs and expectations For instance, a provider might advertise a service that provides quotes for a given ticker symbol, whereas a requester may look for a service that reports current market prices and stock quotes. Services advertise their profiles with Internet wide *discovery services*, such as Middle Agents [21] and other registries (e.g. UDDI [25]), which then match *service requests* against the advertised

Table 1. Description Properties and Functional Attributes.

Description Properties

`serviceName`	The name of the service.
`intendedPurpose`	A high-level description of what constitutes (typical) successful execution of a service.
`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* involved in the service execution.
`requestedBy`	A sub-property of `role` referring to the service requester.
`providedBy`	A sub-property of `role` referring to the service provider.

Functional Attributes

`geographicRadius`	Geographic scope of the service, either at the global scale (e.g. 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 communication language is used (e.g., KQML, SOAP).
`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*, *Information Services* etc.
`qualityGuarantees`	Guarantees that the service promises to deliver, e.g. guaranteeing to provide a response within 3 minutes, etc.
`qualityRating`	Industry-based ratings, such as the "Dun and Bradstreet Rating" for businesses.

profiles, and identify which services provide the best match[7]. Service requests are constructed as partial service profile descriptions, which can then be matched to the profiles of advertised services stored in the registries using DAML+OIL subsumption relations. Advertisements and requests can differ sharply, in level of detail and in the level of abstraction of the terms used. Matches are generally recognized whenever the service advertised is subsumed by (is a particular case of) the service description requested.

The service representation of DAML-S is much richer than the representation provided by emerging standards such as UDDI or WSDL. UDDI's description of a service does not include any capability description, limiting itself to the name, a pointer to the provider of the service and a port where to access the service. In addition, UDDI allows services to refer to "TModels" that are used to link a service to technical specifications or to classification schemes. Therefore, it is possible to ask UDDI for all the services that have a WSDL scheme, but not for all the services that provide a requested functionality. The WSDL

[7] Despite repeated reference to UDDI, DAML-S, like research in Multi-Agent Systems (e.g., [6, 21, 26]), may be used with a variety of different registries and protocols.

specification defines and formats query interactions with a service, but does not provide a model for the semantics of such exchanges. DAML-S service profiles have similarities with service description languages emerging in the Multi-Agent interaction community such as LARKS and OAA [12, 24]. Those languages, like DAML-S, focus on the representation of what the service does rather than where to find the service. DAML-S improves on those service locating models by taking advantage of DAML+OIL ontologies and its inferential capabilities that greatly enhance the possibility for locating relevant services.

4 Modeling Services as Processes

Web Services are Web-accessible programs or devices. Their operation is described in terms of a process model, which details both the control structure and data flow structure of the service, i.e., the possible steps (typically initiated by messages sent by the client) required to execute a service. The process model comprises subclasses and properties of the ProcessModel class.

The two chief components of the process model are the *Process Ontology* which describes a service in terms of its inputs, outputs, preconditions, effects, and, where appropriate, its component subprocesses; and the *Process Control Ontology* which describes each process in terms of its state, including initial activation, execution, and completion. A version of the Process Ontology is released in the current version of DAML-S and can be used to support automated Web Service invocation, composition and interoperation. The Process Control Ontology, which is useful for automated execution monitoring, has not yet been released. We have also defined an ontology of *resources*, and a simple ontology of *time*; they will be described in other publications.

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 planning languages [8], work in programming languages and distributed systems [16, 17], emerging standards in process modeling and workflow technology such as the NIST's Process Specification Language (PSL) [20] and the Workflow Management Coalition effort[8], work on modeling verb semantics and event structure [18], work in AI on modeling complex actions [11], work in agent communication languages [7, 12] and Multi-Agent infrastructure[23], and finally previous work on action-inspired Web Service markup [15].

The primary kind of entity in the Process Ontology is, unsurprisingly, a *process*. The basic Process class has several associated properties. 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. There can also be any number of preconditions, which must all hold in order for the process to be invoked. Finally, the process

[8] http://www.aiim.org/wfmc

can have any number of effects, which are the side effects in the world that result from execution of the program. Outputs and effects can have conditions associated with them. The range of each of these properties, at the upper ontology level, is THING; that is, left totally unrestricted. For most service applications, more specific range restrictions will be used, together with cardinality restrictions. We anticipate that in many cases the range of properties will be subclasses of the class of well-formed formulae in a logical language whose ontology we can define in DAML+OIL.

In DAML-S, we distinguish between *atomic*, *simple*, and *composite* processes:

1. *Atomic* processes are directly invokable (by exchanging messages with the service), have no subprocesses, and execute in a single step, from the perspective of the service requester. (That is, the requester sends a single message, and receives back a single message, in making use of the service.) Atomic processes must provide a grounding that enables a service requester to construct an invocation message and interpret a response message.
2. *Simple* processes, on the other hand, are not directly invokable and are not associated with a grounding. Like atomic processes, they *can* be conceived as having single-step executions. Simple processes are used as elements of abstract processes; 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.
3. *Composite* processes are decomposable into other (non-composite or composite) processes. Their decompositions are specified using control constructs such as SEQUENCE and IF-THEN-ELSE (Table 2). Decompositions show, among other things, the control structure associated with a composition of processes and the input-output dataflow of the composition.

A COMPOSITEPROCESS must have a *composedOf* property by which the control structure of the composite is indicated, using a 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) of which it is composed. For instance, the control construct, SEQUENCE, has a *components* property that ranges over a PROCESSCOMPONENTLIST (a list whose items are restricted to be PROCESSCOMPONENTS, which are either processes or control constructs). 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.

A process can often be viewed at different levels of granularity, either as a primitive, undecomposable process (the "black box" view) or as a composite process (the "glass box" view). 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.

Table 2. Process Constructs

Construct	**Description**
Sequence	Execute a list of processes in a sequential order
Concurrent	Execute elements of a bag of processes concurrently
Split	Invoke elements of a bag of processes
Split+Join	Invoke elements of a bag of processes and synchronize
Unordered	Execute all processes in a bag in any order
Choice	Choose between alternatives and execute one
If-Then-Else	If specified condition holds, execute "Then", else execute "Else".
Repeat-Until	Iterate execution of a bag of processes *until a condition holds.*
Repeat-While	Iterate execution of a bag of processes *while a condition holds*

The DAML-S ontology provides a set of distinguished classes and properties for describing the content and capabilities of Web Services. The DAML+OIL language in which it is specified has a well-defined semantics; however the expressive power of DAML+OIL is not sufficient to restrict DAML-S to all and only the intended interpretations. Recently, we have developed proposals for both an model-theoretic and an execution semantics for DAML-S descriptions. [19, 1]. One approach provides a model-theoretic semantics by describing the intended interpretation of DAML-S in a more expressive first-order logic language [19]. To provide an operational semantics, the representation is then translated into a distributed operational semantics based on High-Level Petri Nets. This allowed us to determine the complexity of important decision procedures (such as reachability and deadlock) for various subsets of the DAML-S process language. In our other approach [1], we use a functional core language to describe DAML-S constructs. A (concurrent) interleaving strict operational semantics for DAML-S is defined, which provides a formal basis for the DAML-S execution model. Together, these proposals allow us to translate DAML-S specifications into an executable process model that can be used for simulation, verification, and composition of DAML-S-described services.

5 Grounding a Service to a Concrete Realization

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

In DAML-S, the abstract content of a message is specified, implicitly, by the input or output properties of an atomic process. Thus, atomic processes, in addition to specifying the primitive processes from which larger processes are composed, can also be thought of as the communication primitives of an (abstract) process specification.

Concrete messages, however, *are* specified explicitly in a grounding. The central function of a DAML-S grounding is to show how the (abstract) inputs and outputs of an atomic process are to be realized concretely as messages, which carry those inputs and outputs in some specific transmittable format. Industry is a long way towards adopting a concrete message specification. As such, in crafting our DAML-S grounding mechanism, we use Web Services Description Language (WSDL), a particular specification language proposal that is representative of efforts in this area and that has strong industry backing.

WSDL "is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate" [4].

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

5.1 Relationships Between DAML-S and WSDL

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

We emphasize that a DAML-S/WSDL grounding involves a *complementary* use of the two languages, in a way that is in accord with the intentions of the authors of WSDL. Both languages are required for the full specification of a grounding. This is because the two languages do not cover the same conceptual space. As indicated by figure 1, the two languages *do* overlap in the area of providing for the specification of what WSDL calls "abstract types", which in turn are used to characterize the inputs and outputs of services. WSDL, by default, specifies abstract types using XML Schema, whereas DAML-S allows for

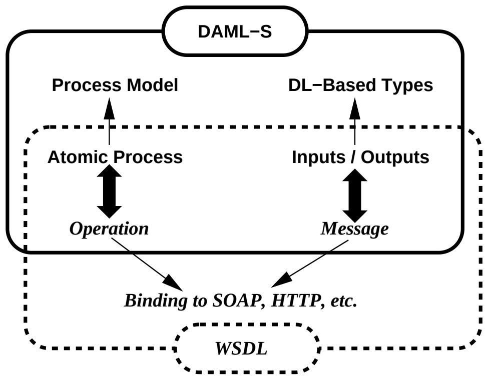

Fig. 1. Mapping between DAML-S and WSDL

the definition of abstract types as (description logic-based) DAML+OIL classes[9]. However, WSDL/XSD is unable to express the semantics of a DAML+OIL class. Similarly, DAML-S has no means, as currently defined, to express the binding information that WSDL captures. Thus, it is natural that a DAML-S/WSDL grounding uses DAML+OIL classes as the abstract types of message parts declared in WSDL, and then relies on WSDL binding constructs to specify the formatting of the messages[10].

A DAML-S/WSDL grounding is based upon the following three correspondences between DAML-S and WSDL. Figure 1 shows the first two of these.

1. A DAML-S atomic process corresponds to a WSDL *operation*. Different types of operations are related to DAML-S processes as follows:
   – An atomic process with both inputs and outputs corresponds to a WSDL *request-response* operation.
   – An atomic process with inputs, but no outputs, corresponds to a WSDL *one-way* operation.
   – An atomic process with outputs, but no inputs, corresponds to a WSDL *notification* operation.
   – A composite process with both outputs and inputs, and with the sending of outputs specified as coming before the reception of inputs, corresponds to WSDL's *solicit-response* operation[11].

[9] XML Schema primitives can also be used to define DAML+OIL properties.

[10] The DAML+OIL classes can either be defined within the WSDL *types* section, or defined in a separate document and referred to from within the WSDL description. In the remainder of this exposition, we describe only the latter approach.

[11] Since a composite process has no grounding, this construct would be grounded indirectly by means of its relationship to a simple process (by the *collapse* property), and hence to an atomic process (by the *realizedBy* property), as mentioned in Section 4. We are considering whether to create a new kind of atomic process in DAML-S, which corresponds directly to the solicit-response operation.

2. The set of inputs and the set of outputs of a DAML-S atomic process each corresponds to WSDL's concept of *message*. More precisely, DAML-S inputs correspond to the parts of an input message of a WSDL operation, and DAML-S outputs correspond to the parts of an output message of a WSDL operation.
   Note that WSDL allows (at most) one input, and (at most) one output message to be associated with an operation. This is in accord with a decision made independently, in DAML-S, that a grounding must map all inputs to (at most) a single message, and similarly for outputs.
3. The types (DAML+OIL classes) of the inputs and outputs of a DAML-S atomic process correspond to WSDL's extensible notion of *abstract type* (and, as such, may be used in WSDL specifications of message parts).

The job of a DAML-S/WSDL grounding is first, to define, in WSDL, the messages and operations by which an atomic process may be accessed, and then, to specify correspondences (1) and (2). Although it is not logically necessary to do so, we believe it will be useful to specify these correspondences both in WSDL and in DAML-S. Thus, as indicated in the following, we allow for constructs in both languages for this purpose.

5.2 Grounding DAML-S Services With WSDL and SOAP

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

Grounding DAML-S with WSDL and SOAP involves the construction of a WSDL service description with all the usual parts (*message, operation, port type, binding*, and *service* constructs), except that the *types* element can normally be omitted. DAML-S extensions are introduced as follows:

1. In each *part* of the WSDL *message* definition, the *daml-property* attribute is used to indicate the fully-qualified name of the DAML-S input or output property, to which this part of the message corresponds. From the property name, the appropriate DAML range class – the class of object which this message part will contain – can easily be obtained.
2. In each WSDL *operation* element, the *daml-s-process* attribute is used to indicate the name of the DAML-S atomic process, to which the operation corresponds.
3. Within the WSDL *binding* element, the *encodingStyle* attribute is given a value such as
   "http://www.daml.org/2001/03/daml+oil.daml", to indicate that the message parts will be serialized in the normal way for class instances of the given types, for the specified version of DAML.

Having completed the WSDL service description, a WsdlGrounding object is constructed (in the DAML-S specification), which refers to specific elements within the WSDL specification, using the following properties:

- *wsdlReference*: A URI that indicates the version of WSDL in use.
- *otherReferences*: A list of URIs indicating other relevant standards employed by the WSDL code (e.g., SOAP, HTTP, MIME).
- *wsdlDocuments*: A list of the URIs of the WSDL document(s) that give the grounding.
- *wsdlOperation*: The URI of the WSDL operation corresponding to the given atomic process.
- *wsdlInputMessage*: An object containing the URI of the WSDL message definition that carries the inputs of the given atomic process, and a list of mapping pairs, which indicate the correspondence between particular DAML-S input properties and particular WSDL message parts.
- *wsdlOutputMessage*: Similar to *wsdlInputMessage*, but for outputs.

6 A Short Walk Through DAML-S

In this final section, we walk through a small DAML-S example[12]. Here we restrict ourselves to illustrating some aspects of the process model and how they relate to the service grounding. Our walk-through utilizes the example of a fictitious book-buying service, CongoBuy. This service is actually a collection of smaller Congo programs (e.g., LocateBook, PutInCart, etc.), each Web-accessible and composed together to form the CongoBuy program.

For a complete specification of DAML-S, please refer to the DAML-S reference document[13]. DAML-S comprises several ontologies in the DAML+OIL (March 2001) markup language. Throughout this example, we will refer to the profile ontology[14] and the process ontology[15]. These ontologies define classes and properties that form the foundation of a service description. To describe a particular service, we specialize these classes and properties by creating subclasses and subproperties specific to the service.

Step 1: Describe Individual Programs The first step in marking up a Web Service is to describe the individual programs that comprise the service. It is the process model that provides a declarative description of a program's properties. The process model conceives each program as either an atomic process, simple or composite process. A non-decomposable Web-accessible program is described as an atomic process. An atomic process is characterized by its ability to be executed by a single (e.g., http) call, that returns a response.

An example of an atomic process is the LocateBook program that takes as input the name of a book and returns a description of the book and its price, if the book is in Congo's catalogue. The simplest way to proclaim `LocateBook` an atomic process is using the `subClassOf` construct as follows.

[12] A more detailed example can be found at http://www.daml.org/services.
[13] http://www.daml.org/services/daml-s/2001/10/daml-s.html
[14] http://www.daml.org/services/daml-s/2001/10/Profile.daml
[15] http://www.daml.org/services/daml-s/2001/10/Process.daml

```
<daml:Class rdf:ID="LocateBook">
  <rdfs:subClassOf rdf:resource="&process;#AtomicProcess"/>
</daml:Class>
```

Associated with each process is a set of properties. Using a program or function metaphor, a process has parameters to which it is associated. Two types of parameters are the DAML-S properties input and (conditional) output, which are defined in the process ontology.

An example of an input for `LocateBook` might be the name of the book. We proclaim this using the `subPropertyOf` construct.

```
<rdf:Property rdf:ID="bookName">
  <rdfs:subPropertyOf rdf:resource="&process;#input"/>
  <rdfs:domain rdf:resource="#LocateBook"/>
  <rdfs:range rdf:resource="&xsd;#string"/>
</rdf:Property>
```

Inputs can be mandatory or optional. In contrast, outputs are generally conditional. For example, when you search for a book in the Congo catalogue, the output may be a detailed description of the book, if Congo carries it, or it may be a "Sorry we don't carry." message. Such outputs are characterized as conditional outputs. To describe a conditional output, the range of output is a class called `ConditionalOutput`, which is a subclass of `Thing`. `ConditionalOutput` in turn has two properties: the condition `coCondition`, and the output `coOutput`. An unconditional output has a zero cardinality restriction on its condition. An example of a conditional output is bookDescription, which is an output conditional upon the book being in the Congo catalogue. If the book is not in Congo's catalogues, then the output is a message to this effect[16].

As above, we can proclaim the conditional outputs of `LocateBook` by specializing our process ontology using `subClassOf` and `subPropertyOf`. Rather than provide the markup here, we illustrate the relations in Figure 6.

The designation of inputs and outputs enables the programs/services that we are describing in DAML-S to be used for automated Web Service invocation. In order to enable the programs/services to be used for automated service composition, we must additionally describe the side-effects of the programs, if any exist. To this end, me must describe the precondition and (conditional) effect properties of our program. They are described analogously to inputs and outputs.

Step 2: Describe the Grounding for Each Atomic Process

Here, we relate `LocateBook` to its grounding, `LocateBookGrounding`. Since `LocateBook` is a class, we need to say: "Every instance (i.e., invocation, or use) of this class has an instance of the `hasGrounding` property, with value `LocateBookGrounding`." The `hasGrounding` property is defined in `Process.daml`.

```
<daml:Class rdf:about="LocateBook">
  <daml:sameClassAs>
```

[16] For many nontrivial applications, the range of the output will be restricted to subclasses of logical well-formed formulae.

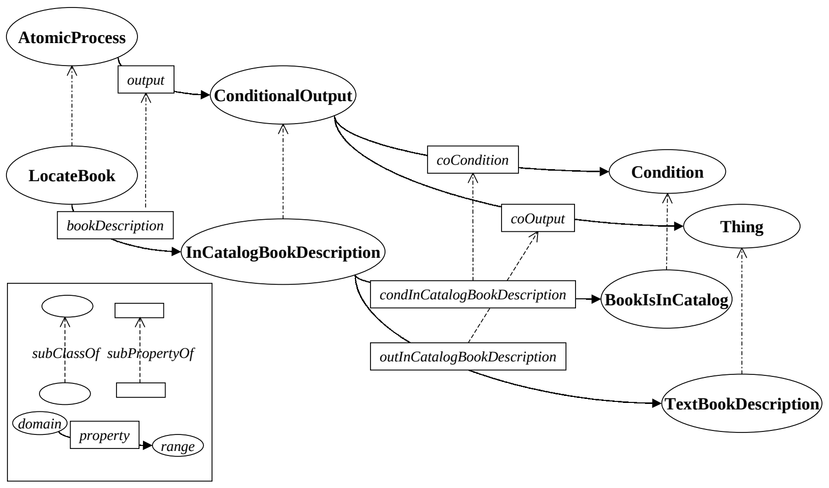

Fig. 2. A Conditional Ouput for LocateBook in DAML-S

```
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#hasGrounding"/>
      <daml:hasValue rdf:resource="#LocateBookGrounding"/>
    </daml:Restriction>
  </daml:sameClassAs>
</daml:Class>
```

The following is an example of a DAML-S Grounding Instance. The "example.com" URIs (`...#FindBook`, `...#LocateBookInput`, etc.) refer to constructs in the corresponding WSDL document (not shown here).

```
<grounding:WsdlGrounding rdf:ID="LocateBookGrounding">
  <grounding:wsdlReference rdf:resource="http://www.w3.org/TR/2001/NOTE-wsdl-20010315">
  <grounding:otherReferences rdf:parseType="daml:collection">
    "http://www.w3.org/TR/2001/NOTE-wsdl-20010315"
    "http://schemas.xmlsoap.org/wsdl/soap/"
    "http://schemas.xmlsoap.org/soap/http/"
  </grounding:otherReferences>
  <grounding:wsdlDocuments rdf:parseType="daml:collection">
    "http://example.com/congo/congobuy.wsdl"
  </grounding:wsdlDocuments>
  <grounding:wsdlOperation
       rdf:resource="http://example.com//locatebook.wsdl#FindBook"/>
  <grounding:wsdlInputMessage
       rdf:resource="http://example.com/locatebook.wsdl#LocateBookInput"/>
  <grounding:wsdlInputMessageParts rdf:parseType="daml:collection">
    <grounding:wsdlMessageMap>
      <grounding:damlsParameter rdf:resource=#bookName>
      <grounding:wsdlMessagePart
          rdf:resource="http://example.com//locatebook.wsdl#BookName">
    </grounding:wsdlMessageMap>
    ... other message map elements ...
  </grounding:wsdlInputMessageParts>
  <grounding:wsdlOutputMessage
           rdf:resource="http://example.com/locatebook.wsdl#LocateBookOutput"/>
  <grounding:wsdlOutputMessageParts rdf:parseType="daml:collection">
    ... similar to wsdlInputMessageParts ...
  </grounding:wsdlOutputMessageParts>
<grounding:WsdlGrounding>
```

Space precludes inclusion of steps 3, 4, and 5 of our walk-through. Step 3 is to describe compositions of the atomic processes. For example, we might describe the composite process `CongoBuyBook` which is a composition of `LocateBook`, `PutInCart`, etc. Step 4 is an optional step, in which we can describe a simple process for our service. Last, but certainly not least is the profile description, which we perform in Step 5. The profile description provides a declarative advertisement for the service. It is partially populated by the process model, if one exists, and this is why it is the last step of our service description.

7 Conclusion

In this paper we have presented DAML-S, an upper ontology for describing Web Services, written in DAML+OIL. Three aspects of DAML-S were presented: the service profile, the process model, and the service grounding (with focus on the last one). Service grounding is critical to the successful deployment of DAML-S, since it provides the connection between our Semantic Web approach and the emerging industry standards for Web Service description (e.g. WSDL), demonstrating that DAML-S is complementary to the mainstream industry efforts.

Acknowledgments

This work has profited from discussions with a number of people, most notably including Jim Hendler. 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 and Nokia Mobile Phones.

References

1. A. Ankolekar, F. Huch and K. Sycara. Concurrent Semantics for the Web Services Specification Language Daml-S. In *Proc. of the Coordination 2002 Conf.*, 2002.
2. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.
3. U. C. Bureau. North american industry classification system (naics). http://www.census.gov/epcd/www/naics.html, 1997.

4. 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.
5. DAML-S Coalition: A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. DAML-S: Semantic markup for Web services. In *Proc SWWS*, pages 411–430, 2001.
6. K. Decker, K. Sycara, and M. Williamson. Middle-agents for the internet. In *IJCAI97*, 1997.
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. M. Ghallab et. al. PDDL-the planning domain definition language v. 2. Tech Report,CVC TR-98-003/DCS TR-1165, Yale University, 1998.
9. J. Hendler and D. L. McGuinness. Darpa Agent Markup Language. *IEEE Intelligent Systems*, 15(6):72–73, 2001.
10. O. Lassila. Serendipitous Interoperability. In E. Hyvönen, editor, *The Semantic Web – Proc. the Kick-Off Seminar in Finland*, To appear, 2002.
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. D. McDermott, M. Burstein, and D. Smith. Overcoming ontology mismatches in transactions with self-describing agents. In *Proc. SWWS*, pages 285–302, 2001.
14. S. McIlraith and T. C. Son. Adapting Golog for composition of Semantic Web services. In *Proc. KR2002*. To appear, 2002.
15. S. McIlraith, T. C. Son, and H. Zeng. Semantic Web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.
16. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
17. R. Milner. Communicating with Mobile Agents: The pi-Calculus. Cambridge University Press, Cambridge, 1999.
18. S. Narayanan. Reasoning about actions in narrative understanding. In *Proc. IJCAI'1999*, pages 350–357. 1999.
19. S. Narayanan and S. McIlraith. Simulation, verification, and automated composition of Web Services. In *Proc. WWW2002*, To appear 2002.
20. 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.
21. M. Paolucci, T. Kawmura, T. Payne and K. Sycara. Semantic Matching of Web Services Capabilities. In *First Int. Semantic Web Conf.*, To appear 2002.
22. T. Payne, R. Singh and K. Sycara. Browsing Schedules - An Agent-based approach to navigating the Semantic Web In *First Int. Semantic Web Conf.*, To appear 2002.
23. K. Sycara and M. Klusch. Brokering and matchmaking for coordination of agent societies: A survey. In *Coordination of Internet Agents*, 2001.
24. K. Sycara, M. Klusch, S. Widoff, and J. Lu. Dynamic service matchmaking among agents in open information environments. *Journal ACM SIGMOD Record*, 28(1):47–53, 1999.
25. UDDI. The UDDI Technical White Paper. http://www.uddi.org/, 2000.
26. H.-C. Wong and K. Sycara. A taxonomy of middle-agents for the internet. In *ICMAS'2000*, 2000.