Mobilizing the Semantic Web with DAML-Enabled Web Services

Sheila A. McIlraith
Knowledge Systems Laboratory
Department of Computer Science
Stanford University
Stanford, CA 94305-9020, USA
1-650-723-7932
sam@ksl.stanford.edu

Tran Cao Son*
MSC CS, PO Box 30001
Department of Computer Science
New Mexico State University
Las Cruces, NM 88001
1-505-646-1930
tson@cs.nmsu.edu

Honglei Zeng
Knowledge Systems Laboratory
Department of Computer Science
Stanford University
Stanford , CA 94305-9020, USA
1-650-723-7932
hlzeng@ksl.stanford.edu

ABSTRACT

The Web is evolving from a repository for text and images to a provider of services – both information-providing services, and services that have some effect on the world. Today's Web was designed primarily for human use. To enable reliable, large-scale automated interoperation of services by computer programs or agents, the properties, capabilities, interfaces and effects of Web services must be understandable to computers. In this paper we propose a vision and a partial realization of precisely this. We propose markup of Web services in the DAML family of semantic Web markup languages. Our markup of Web services enables a wide variety of agent technologies for automated Web service discovery, execution, composition and interoperation. We present one logic-based agent technology for service composition, predicated on the use of reusable, task-specific, high-level generic procedures and user-specific customizing constraints.

1. INTRODUCTION

The Web, once solely a repository for text and images, is evolving into a provider of services – *information-providing services* such as flight information providers, temperature sensors, and cameras; and *world-altering services* such as flight booking programs, sensor controllers, and a variety of E-Commerce and B2B applications. These services are realized by Web-accessible programs, databases, sensors, and by a variety of other physical devices. It is predicted that in the next decade, computers will be ubiquitous, and that most devices will have some sort of computer inside. Vint Cerf, one of the fathers of the Internet, views the Internet's population by smart devices as the harbinger of a new revolution in Internet technology.

Today's Web was designed primarily for human interpretation and human use. Nevertheless, we are seeing increased automation of Web service interoperation, primarily in B2B and E-Commerce applications. Generally, such interoperation is realized through APIs that incorporate hand-coded information extraction code to locate and extract content from the HTML syntax of a Web page presentation layout. Unfortunately, when a Web page changes its presentation layout, the API must be modified to prevent failure. Fundamental to reliable, large-scale interoperation of Web services by computer programs or agents [8] is the need to make Web services directly understandable – to create a Semantic Web [2] of services whose properties, capabilities, interfaces and effects are encoded in an unambiguous, machine-interpretable form.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission by the authors.

Semantic Web Workshop 2001 Hongkong, China

Copyright by the authors.

The realization of the Semantic Web is underway with the development of content markup languages such as XML, SHOE [11], very recently DAML+OIL[1], and eventually DAML-L. The former languages are not sufficiently expressive for many applications, and/or are without sufficient semantics. The latter, inspired by artificial intelligence (AI) knowledge representation languages and still in the midst of development, are more expressive logical languages that build on top of XML and RDF [10] to enable the markup and manipulation of more complex taxonomic and logical relations between entities on the Web.

We believe that a fundamental component of the Semantic Web will be the markup of Web services to make them computer interpretable, use-apparent, and agent-ready. This paper addresses precisely this component. We present an approach to Web service markup that provides an agent-independent *declarative API*, capturing the data and metadata associated with a service together with specifications of its properties and capabilities, the interface for its execution, and the prerequisites and consequences of its use. Markup exploits ontologies to facilitate sharing, reuse, composition, mapping, and succinct local Web service markup. Our vision is partially realized by Web service markup in a dialect of the newly proposed DAML (DARPA Agent Markup Language) [9] family of semantic Web markup languages. Such so-called *semantic markup* of Web services provides a *distributed knowledge base* (KB) that agents can reason over to perform a variety of tasks including automated Web service discovery, execution, composition, and interoperation. To illustrate this point, we present an agent technology based on reusable generic procedures and customizing constraints that exploits and showcases our Web service markup. This agent technology is realized using the first-order language of the situation calculus [15] and an extended version of the agent

[1] http://www.daml.org/2000/10/daml-oil,

* Most of this work was performed while the author was a Postdoctoral Fellow at the Knowledge Systems Lab.

programming language ConGolog [3] together with deductive machinery. In Section 2 we provide a set of motivating Web service tasks and define the basic components of our framework. In Section 3 we present our approach to semantic markup of Web services. In Section 4 we present a logic-based agent technology that exploits the markup we propose to perform automated service composition and interoperation.

Throughout this paper, we illustrate concepts in terms of travel examples. Clearly, there is nothing in this technology that restricts it to this domain. Our work is designed to be domain-independent. Similarly, the vision we present in this paper may be realized by a variety of different semantic Web markup languages and may be exploited by a variety of different agent architectures and technologies. The markup and the agent technology we present here are but one of many possible realizations – one we believe is well justified. Finally, our realization is rapidly evolving as we exploit important ongoing extensions to the DAML family of semantic Web markup languages.

2. SEMANTIC WEB SERVICES

To realize our vision of Semantic Web Services we are a) creating semantic markup of Web services that enables them to be computer-interpretable, use-apparent, and agent-ready; and b) developing agent technology that exploits this semantic markup to support automated interoperability of Web services. Driving the development of our markup and agent technology are the automation tasks that semantic markup of Web services will enable. We contrast the following three tasks.

1. Automatic Web service discovery.

E.g., Find me a service that sells airline tickets between San Francisco and Toronto, and that accepts payment by Diner's Club credit card.

Automatic Web service discovery involves the automatic location of Web services that provide a particular service and that adhere to requested properties. Currently, this task must be performed by a human who may use a search engine to find a service, and who may then read the Web page associated with that service, or execute the service manually, to see whether it adheres to the requested properties. With semantic markup of services, the information necessary for Web service discovery can be specified as computer-interpretable semantic markup at the service Web sites, and a service registry or (ontology-enhanced) search engine used to automatically locate appropriate services.

2. Automatic Web service execution.

E.g., Buy me an airline ticket from www.acmetravel.com *on UAL flight 1234 from San Francisco to Toronto on March 3.*

Automatic Web service execution involves the automatic execution of an identified Web service by a computer program or agent. To execute a particular service on today's Web, such as buying an airline ticket, generally, 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 encoded. In either case, human interpretation is required to understand what information is required to execute the service, and to interpret the information returned by the service. Automatic execution of a Web service can be thought of as a function call or a call to a process of functions. Semantic markup of Web services provides a declarative, computer-interpretable API for executing these function/service calls, that an agent can interpret to understand what input is necessary to the service call, what information will be returned and how to execute the service automatically.

3. Automatic Web service composition and interoperation.

E.g., Make the travel arrangements for my WWW10 conference.

As the name implies, this task involves the automatic selection, composition and interoperation of appropriate Web services to perform some task, given a high-level description of the objective of the task. Currently, if some task requires a composition of Web services that must interoperate, then the user must select the Web services, manually specify the composition, ensure that any software for interoperation is custom-created, and provide the input at choice points (e.g., selecting a flight). With semantic markup of Web services, the information necessary to select, compose, and respond to services is encoded at the service Web sites. Software can be written to manipulate this markup, together with a specification of the objectives of the task, to achieve the task automatically. Service composition and interoperation leverage automatic discovery and execution.

Of the three tasks listed above, none is entirely realizable with today's Web, primarily because of lack of content markup and a suitable markup language. Academic research on Web service discovery is growing out of agent match-making research such as the Lark system [16], which proposes attributes and a representation for annotating agent capabilities so that they can be located and brokered. Recent industrial efforts have been focusing primarily on improving Web service discovery and aspects of service execution through initiatives such as the Universal Description, Discovery and Integration (UDDI) standard service registry; the XML-based Web Service Description Language (WSDL) released in September 2000 as a framework independent Web service description language; and ebXML, an initiative of the United Nations (UN/CEFACT) and OASIS to standardize a framework for trading partner interchange. E-business infrastructure companies are now beginning to announce platforms to support some level of Web-service automation. Examples of such products include Hewlett-Packard's e-speak, a description, registration, and dynamic discovery platform for e-services; Microsoft's .NET and BizTalk tools; Oracle's Dynamic Services Framework; IBM's Application Framework for E-Business; and Sun's Open Network Environment (ONE). VerticalNet Solutions, anticipating and wishing to accelerate the markup of services for discovery, is building ontologies and tools to organize and customize Web service discovery, and with their OSM Platform, delivering an infrastructure that coordinates Web services for public and private trading exchanges.

What distinguishes our work in this arena is our semantic markup of Web services in an expressive semantic Web markup language with a well-defined semantics. The semantic markup presented in this paper provides a semantic layer that should comfortably sit

on top of efforts such as WSDL, enabling a richer level of description and hence more sophisticated interactions and reasoning at the agent/application level. As a demonstration of this claim, we present agent technology that performs *automatic* Web service composition, an area that industry is not yet tackling in any great measure.

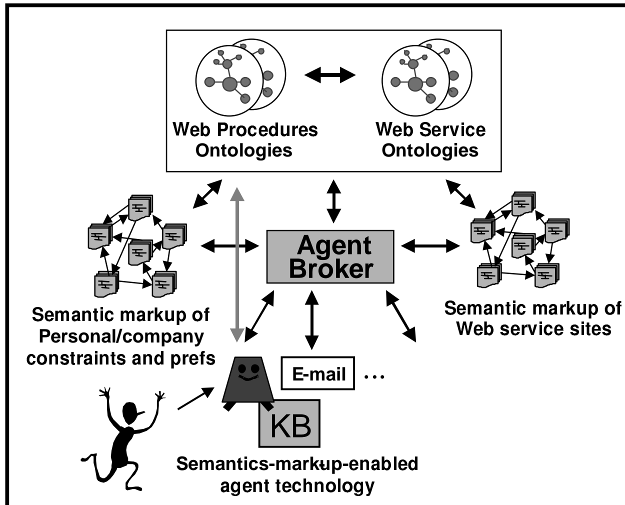

Figure 1. A Framework for Semantic Web

Figure 1 illustrates the basic components of our Semantic Web Services framework. It is composed of semantic markup of Web services, user constraints, and Web agent procedures. This markup leverages ontologies of defined terms and concepts that are stored in reusable and sharable Web service ontologies and Web agent procedures ontologies. The semantic markup of Web services collectively forms a distributed KB of Web services that can be accessed and reasoned about. In so doing, it provides a means for agents to populate their local KBs so that they can reason about Web service properties and capabilities in order to accomplish some of the automated reasoning tasks identified above. In addition to the markup, our framework includes a variety of agent technologies each of which communicates with the Web services via an agent broker that sends requests for services to appropriate Web services, and dispatch responses back to the agent. These brokers will eventually exploit both local markup and Web service ontologies to assist in locating appropriate service providers either internally or through the use of an automatic discovery agent. Key components of our Semantic Web Services will be described in further detail in the sections to follow.

3. SEMANTIC WEB SERVICE MARKUP

The three tasks described in Section 2 serve as drivers for the development of our semantic Web services markup. We are marking up 1) Web services such as Yahoo's™ driving direction service, or United Airlines'™ flight schedule information service; 2) user and group constraints and preference such as user Bob's schedule, that he prefers to drive if the driving time to his destination is less than 3 hours, that he likes to get stock quotes from the E*Trade™ Web service, etc.; and 3) agent procedures which are (partial) compositions of existing Web services, designed to perform a particular task, and marked up for sharing and reuse by other users. Examples of such procedures include Bob's company's business travel booking procedure, or Bob's friend's stock assessment procedure. In the subsection that follows, we make the case for the DAML family of markup languages we are using in our work. Following this, we discuss further details of our approach to semantic Web service markup.

3.1 Why DAML?

In recent years, a number of markup languages have been developed with a view to creating languages that are adequate for realizing the Semantic Web. The construction of these languages is evolving according to a layered approach to language development as depicted in Figure 2, which was modified from [6].

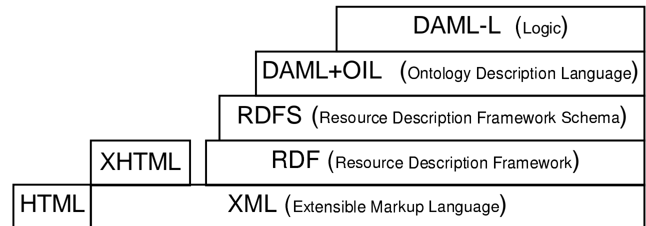

Figure 2. Layered Language Development

XML was the first language to separate the markup of Web content from Web presentation, facilitating the representation of task- and domain-specific data on the Web. Unfortunately, XML lacks semantics. As such, computer programs cannot be guaranteed to determine the intended interpretation of XML tags. For example, a computer program would not be able to identify that `<SALARY>` data refers to the same information as `<WAGE>` data, or that the `<DUE-DATE>` specified at a Web service vendor's site may be different from the `<DUE-DATE>` at the purchaser's site.

The Resource Description Framework (RDF) was developed by the World Wide Web Consortium (W3C) as a standard for metadata, with the goal of adding a formal semantics to the Web. RDF is defined on top of XML to provide a data model and syntax convention for representing the semantics of data in a standardized interoperable manner. It provides a means of describing the relationships among resources (basically anything namable by a URI) in terms of named properties and values. The RDF working group also developed RDF Schema (RDFS), an object-oriented type system that can be effectively thought of as a minimal ontology modeling language. Although RDF and RDFS provide good building blocks for defining a Semantic Web markup language, they lack expressive power. For example, one cannot define properties of properties, necessary and sufficient conditions for class membership, equivalence and disjointness of classes, and the only constraints expressible are domain/range constraints on properties. Finally, and perhaps most importantly, the semantics remains under-specified.

Recently, there have been several efforts to build upon RDF(S) with more AI-inspired knowledge representation languages such as SHOE [11], DAML-ONT [8], OIL [17], and most recently

DAML+OIL. DAML+OIL is the second in, what we refer to as, the DAML family of markup languages, replacing DAML-ONT as an expressive ontology description language for markup. Building on top of RDF(S) and with its roots in AI description logics, DAML+OIL overcomes many of the expressiveness inadequacies plaguing RDFS, and most importantly, has a well defined model-theoretic semantics as well as an axiomatic specification that determines the intended interpretations of the language. DAML+OIL is unambiguously computer-interpretable, thus making it amenable to agent interoperability and automated reasoning techniques, such as those we exploit in our agent technology.

In the next half year, DAML will be extended with the addition of DAML-L, a logical language with a well-defined semantics and the ability to express at least propositional Horn clauses. Horn clauses enable compact representation of constraints and rules for reasoning. Consider an airline flight information service that wishes to encode whether a flight shows a movie. One way to do this is to create markup for each flight (such as a database) indicating whether or not the flight has a movie. A more compact representation is to write the constraint `flight-over-3-hours` → `movie` and to use deductive reasoning to infer whether a flight has a movie. This representation is more compact, informative, and easier to modify than an explicit enumeration of whether each individual flight has a movie. Similarly, such clauses can be used to represent business rules in a compact form.

DAML+OIL and DAML-L together will provide a markup language for the Semantic Web with reasonable expressive power and a well-defined semantics. Should further expressive power be necessary, the layered approach to language development allows a more expressive logical language to extend DAML-L, or to act as an alternate extension to DAML+OIL. In the subsection to follow we discuss our approach to markup of semantic Web services using the DAML family of languages. Since DAML-L has not yet been developed, current markup is in a combination of DAML+OIL and a subset of first-order logic. Our markup evolves as the DAML family of languages evolves.

3.2 DAML markup for Web services

In this section we discuss, in succession, our approach to the DAML markup of Web services, and our approach to DAML markup of user constraints and preferences. Our approach to the DAML markup of agent procedures can be realized by a similar set of concepts to those used for Web services.

Our DAML markup provides a declarative representation of Web service and user constraint knowledge. A key feature of our markup is the exploitation of ontologies, which are supported by DAML+OIL's roots in description logics and frame systems. We use ontologies to encode the classes and subclasses of general concepts and relations pertaining to services and user constraints. (E.g., `BuyUALTicket` and `BuyLufthansaTicket` are subclasses of the service `BuyAirlineTicket`, inheriting the parameters `customer`, `origin`, `destination`, etc.). General Web service ontologies are augmented by domain-specific ontologies that inherit concepts from the general ontologies and that additionally encode concepts that are specific to the individual Web service or user.

The use of ontologies enables the *sharing* of common concepts, the *specialization* of these concepts and vocabulary for *reuse* across multiple applications, the *mapping* of concepts between different ontologies, and the *composition* of new concepts from multiple ontologies. Ontologies support the development of *succinct service/user-specific markup* by enabling an individual service/user to inherit much of its semantic markup from ontologies, thus requiring only minimal instantiation or specialization at the Web site. Most importantly, the use of ontologies provides a means of giving semantics to markup by constraining or grounding its interpretation. Web services and users need not exploit Web service ontologies, but we foresee many instances where communities will wish to agree upon a standard definition of terminology, and to encode it in an ontology.

3.2.1 Markup of Web services

Our DAML markup of Web services is driven by the three automation tasks identified in Section 2: discovery, execution, and composition and interoperation. Collectively our markup provides

- declarative advertisements of service properties and capabilities which can be used for automatic service discovery;
- declarative APIs for individual Web services that are necessary for automatic Web service execution; and
- declarative specifications of the prerequisites and consequences of individual service use that are necessary for automatic service composition and interoperation.

The semantic markup of multiple Web services collectively forms a distributed knowledge base of Web services that can be accessed and reasoned about. Semantic markup can be used to populate detailed registries of the properties and capabilities of Web services for knowledge-based indexing and retrieval of Web services by agent brokers and human beings alike. It can also provide a way for agents to interactively populate their knowledge bases so that they can reason about Web services. Marked up Web services will be computer-interpretable, use-apparent, and agent-ready.

Our Web service markup comprises a number of different ontologies that provide the backbone for our Web service descriptions. We define the class of services, Service, and divide it into two subclasses `PrimitiveService` and `ComplexService`. In the context of the Web, a primitive service is an individual Web-executable computer program/sensor/device that does not call another Web service. There is no ongoing interaction between the user and a primitive service. The service is called and a response is returned. An example of a primitive service is a Web-accessible program that returns a postal code, given a valid address. In contrast, a complex service is composed of multiple primitive services, often requiring an interaction or conversation between the user and the services, so that the user can make choices. An example might be the interaction with www.amazon.com to buy a book.

Domain–specific Web service ontologies are built as subclasses of these general classes. For example, we may define the class `Buy`, with subclass `BuyTicket`, which has subclasses `BuyMovieTicket`, `BuyAirlineTicket`, etc.

BuyAirlineTichet has subclasses BuyUALTicket, BuyLufthansaTicket, and so on. Each service is either a PrimitiveService or a ComplexService. Associated with each service is a set of Parameters. For example, the class Buy will have parameter Customer. BuyAirlineTicket will inherit the Customer parameter and will also have parameters Origin, Destination, DepartureDate, etc. Domain-specific ontologies are also constructed to describe parameter values. For example, the values of Origin and Destination are restricted to instances of the class Airport. BuyUALTicket inherits these parameters, further restricting them to Aiports whose property Airlines includes UAL. These value restrictions provide an important way of describing Web service properties, which supports better brokering of services, simple type checking for our declarative APIs, and which we have used in our agent technology to create customized user interfaces.

Also associated with an individual service are other properties of the service, some of which may be inherited. In the case of BuyUALTicket these would include the service URL, CompanyName, IntendedUse, valid methods of payment, travel bonus plans accepted, etc. This markup provides a declarative advertisement of service properties and capabilities, which is computer interpretable and can be used for automatic service discovery. Note that much of the markup can exist in reusable ontologies, if the service provider so chooses. In such a case, the markup at the individual Web site specifies which ontologies it is using and adds additional site-specific markup.

3.2.1.1 Markup for Web service execution

To automate Web service execution, markup must enable a computer agent to automatically construct and execute a Web service request, and interpret and potentially respond to the service's response. Markup for execution requires a dataflow model and we use both a *function metaphor* and a *process or conversation model* to realize our markup. Each primitive service is conceived as a function with Input values and potentially multiple alternative Output values, conditioned on state. For example if the user orders a book, the response will be different depending upon whether the book is in stock, out of stock, or out of print. Complex services are conceived as a composition of functions (services) whose output may require an exchange of information between the agent and individual services. For example, a complex service that books a flight for a user may involve first finding flights that meet the user's request, then suspending until the user selects one flight, to complete the service. Complex services are composed of primitive or complex services using typical programming languages and business process modeling language constructs such as Sequence, Iteration, If-the-Else, While loop. This markup provides declarative APIs for individual Web services that are necessary for automatic Web service execution. It additionally provides a process dataflow model for complex services. In order for an agent to respond automatically to a complex service execution, i.e., to automatically interoperate with that service, it will require some of the information encoded for automatic composition and interoperation.

3.2.1.2 Markup for Web service composition

The function metaphor used for automatic Web service execution markup provides information about data flow, but it does not provide information about *what* the Web service actually does. In order to automate service composition and in order for services/agents to interoperate, we must also encode the effects a service has upon the world. For example, when a human being goes to www.amazon.com and successfully executes the BuyBook 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 markup, nor part of the function-based specification provided for automatic execution. In order to automate Web service composition and interoperation, or even to select an individual service to meet some objective, prerequisites and consequences of Web service execution must be encoded for computer use.

Our DAML markup of Web services for automatic composition and interoperability is built upon an AI-based *action metaphor*. We conceive each Web service as an Action – either a PrimitiveAction, or a ComplexAction. Primitive services are conceived as primitive actions, complex services as complex actions. Primitive actions are in turn conceived as either world-altering actions that change the state of the world, such as debiting the user's credit card, booking the user a ticket, etc.; information-gathering actions that change the agent's state of knowledge, so that after execution of the action the agent knows a piece of information; or some combination of the two.

An advantage of exploiting an action metaphor to describe Web services is that it enables us to bring to bear the vast AI research on reasoning about action, to support automated reasoning tasks such as Web service composition. However, in developing our markup we choose to remain agnostic with respect to an action representation formalism. In the AI community, there is widespread disagreement over the best action representation formalism. As a consequence, different agents use very different internal representations for reasoning about, and planning sequences of actions. The planning community has addressed this lack of consensus by developing a specification language for describing planning domains – Plan Domain Description Language (PDDL) [7]. We adopt this language here, specifying each of our Web services in terms of PDDL-inspired Parameters, Preconditions and Effects. The Input and Output necessary for automatic Web service execution also play the role of KnowledgePreconditions and KnowledgeEffects for the purposes of Web service composition and interoperation. We assume, as in the planning community, that users will compile this general representation into an action formalism that best suits their reasoning needs. Translators already exist from PDDL to a variety of different AI action formalisms.

ComplexActions, like complex services, are compositions of individual services, however dependencies between these compositions are often predicated on the state of the world (preconditions and effects) rather than on data (the input and output of services) as is the case with the execution-motivated markup. ComplexActions are composed of PrimitiveActions or other ComplexActions using

typical programming languages and business process modeling languages constructs such as `Sequence`, `Parallel`, `If-then-else`, `Iteration`, `While`, etc.

This completes our discussion of our approach to DAML markup of Web service. The other markup we provide is of user constraints and preferences.

3.2.2 DAML markup of user constraints

Our vision is that agents will exploit users' constraints and preferences to help customize users' requests for automatic Web service discovery, execution, or composition and interoperation. Examples of user constraints and preferences include user Bob's schedule, his travel bonus point plans, that he prefers to drive if the driving time to his destination is less than 3 hours, that he likes to get stock quotes from the E*Trade™ Web service, that his company requires that all domestic business travel must be on a particular air carrier, that business travel must be approved, etc. The actual markup of user constraints is relatively straightforward, given DAML-L. Most constraints can be expressed as formulae in Horn clause logic. What is more challenging is the agent technology to appropriately exploit this markup. In the section that follows we will see one example of such an agent technology.

Briefly, each user has a set of constraints whose vocabulary is defined in Web services ontologies or in the ontologies of the agent procedures. Inheriting terminology from these ontologies ensures, for example, that Bob's constraint about `DrivingTime` is enforced by determining the value of `DrivingTime` from a service that uses the same notion of `DrivingTime`. User constraints can also be inherited from an ontology of group constraints. For example, the employees of a company may have certain constraints with respect to the services their company uses or trusts, restrictions on what air carriers they may use, what authorization they require to execute certain transactions. All this information can be encoded in an ontology that individual users can inherit.

This concludes our discussion of DAML Markup for Semantic Web Services. Our objective was to describe our basic approach and the rationale behind it. Further details on our DAML markup will appear in another publication or at our Web site. The Web service markup that we describe here is being merged with markup developed by SRI as part of an effort to provide a unified core markup language for Web Services [4]. The resultant language, DAML-S will be a joint effort between Stanford Knowledge Systems Lab, SRI, CMU, BBN and Nokia. In the section to follow we describe an agent technology we have developed that exploits our DAML markup to perform Web service composition.

4. DAML-ENABLED AGENTS

Our semantic markup of Web services enables a wide variety of agent technologies. In this paper we present one agent technology we are developing that exploits DAML markup of Web services to perform automated Web service composition.

Consider the example task given in Section 2, "Make my travel arrangements for my WWW10 conference." If you were to perform this task yourself using services available on the Web, you might first find the WWW10 conference Web page and determine the location and dates of the conference. Based on the location, you would decide upon the most appropriate mode of transportation. If traveling by air, you might then check flight schedules with one or more Web services, book flights, and arrange transportation to the airport through another Web service. Otherwise, you might book a rental car. You would then need to arrange transportation and accommodations at the conference location, and so on.

While the procedure is lengthy and somewhat tedious to perform, *what* you have to do to make travel arrangements is easily described by the average person. Nevertheless, many of us know it is very difficult to get someone else to actually make your travel arrangements for you. What makes this task difficult to perform is not the basic steps but the need to make choices and customize the procedure in order to enforce your constraints. For example, you may only like to fly on selected airlines where you can collect travel bonus points. You may have scheduled activities that you need to work around. You may or may not prefer to have a car at the conference location, which will affect the selection of hotel. In turn, your company may require that you get approval for travel, they may place budget constraints on your travel, or they may mandate that you use particular air carriers. Constraints can be numerous and consequently difficult for another human being to keep in their head and to simultaneously satisfy. Fortunately, the enforcement of complex constraints is something a computer does well.

Our objective is to develop agent technology that will perform these types of tasks *automatically* by exploiting DAML markup of Web services and DAML markup of user constraints. Clearly we cannot simply program an agent to perform these tasks because there is so much variability in how a task is achieved, depending upon an individual's constraints. One approach to addressing the problem would be to exploit AI techniques for planning, and to build an agent that plans a sequence of Web service requests to achieve the goals of the task, predicated on the user's constraints. This is certainly a reasonable approach, but planning is computationally intensive and in this case, we actually know much of what the agent needs to do. We just need the agent to fill in the details and manage the traveler's constraints.

We argue that many of the activities a user may wish to perform on the semantic Web, within the context of his/her place of work or at home, can be viewed as customizations of reusable, high-level *generic procedures*. Our vision is to construct such reusable, high-level generic procedures, to represent them as distinguished services in DAML using a subset of the markup structure used for complex services, and to archive them in sharable generic procedures ontologies so that multiple users can access them. A user could then select a task-specific generic procedure from the ontology and submit it to their DAML-enabled agent. The agent would automatically customize the procedure with respect to the user's DAML encoded constraints, the current state of the world, and available services, to generate and execute a sequence of requests to Web services to perform the task. This is the vision our agent technology is realizing.

4.1 Generic Procedures and User Constraints

Our approach is logic based, affording us many of the attributes of logic. This includes compact representations of concepts and relations, the ability to reason about actions and procedures in our

language using deductive machinery, verification of certain properties of our procedures, and the potential to provide soundness and completeness guarantees with respect to any reasoning we do. We first overview our logic-based technology to realize generic procedures and customizing constraints, and then we describe how it fits into the framework in Figure 1 to communicate with the semantic Web.

We build on our research on model-based programming (e.g., [13]) and on research into the agent programming language Golog and its variants (e.g., [3]) to provide a DAML-enabled agent programming capability that supports writing generic procedures to performing Web service-based tasks. Model-based programs comprise a *model*, which in this case is the agent's KB, and a *program*, in this case the generic procedure we wish to execute. We argue that the situation calculus, a logical language for reasoning about action and change, and ConGolog (e.g.,[3]), an agent programming language built on top of the situation calculus, provide a compeling language for realizing our agent technology. When a user requests a generic procedure, such as a generic travel arrangements procedure, the agent populates its local KB with the subset of the PDDL-inspired DAML Web service markup that is relevant to the procedure. It also adds the user's constraints to its KB. Exploiting our action metaphor for Web services, the agent KB provides a logical encoding of the preconditions and effects of the Web service "actions" in the language of the situation calculus.

Model-based programs, such as our generic procedures, are written in the agent programming language ConGolog, without prior knowledge of what specific services the agent will use or of how exactly to use the available services. As such, they capture *what* to do, but not exactly *how*. They use procedural programming language constructs (if-then-else, while, etc.) composed with concepts defined in our DAML service and constraints ontologies to describe the procedure. The agent's model-based program is not executable as is. It must be deductively instantiated in the context of the agent's KB, which includes properties of the agent and its user, properties of the specific services we are utilizing, and the state of the world. The instantiation is performed by deductive machinery. Instantiated programs are simply sequences of primitive actions (individual Web services), which are sent to the agent broker as requests for service. The great advantage of these generic procedures, as we illustrate in the example below, is that the same generic procedure, called with different parameters and user constraints, can generate very different sequences of actions.

4.2 ConGolog

We argue that an augmented version of the logic programming language, ConGolog and the deductive machinery of Prolog provide a natural formalism for realizing our agent technology. In this section, we introduce ConGolog and describe its use for Web service composition. ConGolog is built on top of the situation calculus, a first-order logical language for reasoning about action and change. We begin with a short introduction to the situation calculus.

In the situation calculus, the state of the world is expressed in terms of functions and relations (so-called fluents) that are true/false or have a particular value in a situation, *s* (e.g., flightAvailable(origin,dest, date,s). A situation *s* is a history of the actions (e.g., *a*) performed from an initial, distinguished situation S_0. The function *do(a,s)* maps a situation and action into a new situation. A situation calculus theory, *KB* comprises the following sets of axioms (See [15] for details):

- foundational axioms of the situation calculus, Σ
- successor state axioms, KB_{SS}
- action precondition axioms, KB_{AP}
- axioms describing the initial situation, KB_{S_0}
- unique names for actions, KB_{UNA}
- domain closure axioms for actions, KB_{DCA}[2]

The world is conceived as a tree of situations, starting at an initial situation S_0, and evolving to a new situation through the performance of an action *a* (e.g., BuyUALTicket(origin, dest, date)). Figure 3 illustrates the tree of situations induced by a situation calculus theory with actions $a_1, \ldots, a_n$ (ignore the X's for the time being). The tree is not actually computed, but it reflects the search space induced by the situation calculus KB. Were we to choose to do so, we could perform deductive plan synthesis to plan sequences of Web service "actions" over this search space. Instead, we develop generic procedures in ConGolog.

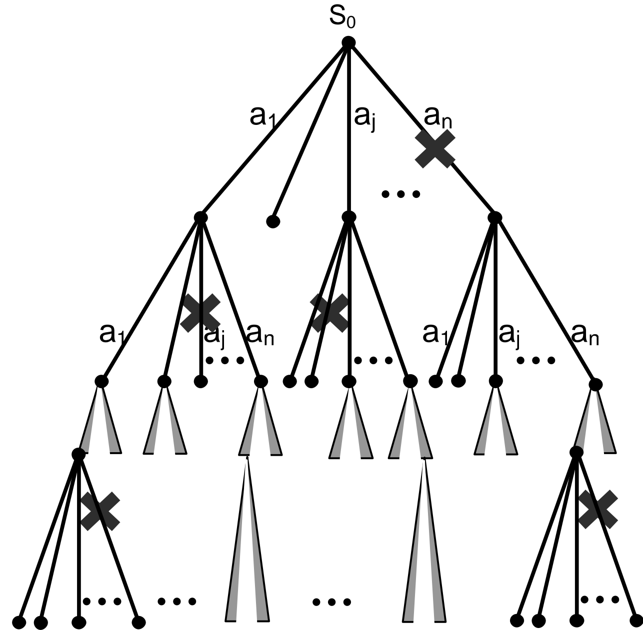

Figure 3. The Tree of Situations

ConGolog is a high-level logic programming language developed at the University of Toronto (e.g., [3]). Its primary use is for robot programming and to support high-level robot task planning, but it has also been used for agent-based programming (e.g., meeting scheduling).

[2] Not always necessary, but we will require it here.

ConGolog provides a set of extralogical constructs (if-then-else, while) for assembling primitive and complex situation calculus actions into other complex actions. We leave detailed discussion of ConGolog and its variants to (e.g., [3]) and simply describe the basic constructs of the ConGolog language. Let δ_1 and δ_2 be complex actions and let φ and a be so-called pseudo fluents and pseudo actions, respectively, i.e., a fluent/action in the language of the situation calculus with all its situation arguments suppressed. The following is a subset of the constructs in the ConGolog language.

primitive action -- a
test of truth -- $\varphi?$
sequence – $(\delta_1;\delta_2)$
nondeterministic choice between actions – $(\delta_1|\delta_2)$
nondeterministic choice of arguments -- $\pi x.\delta$
iteration -- $\delta*$
conditional – **if** φ **then** δ_1 **else** δ_2 **endIf**
loop -- **while** φ **do** δ **endWhile**
procedure -- **proc** $P(v)$ δ **endProc**

These constructs can be used to write generic procedures, which are themselves complex actions in ConGolog. The instruction set for these complex actions, is simply the general Web services (e.g. BookAirlineTicket) or other complex actions. The following are examples of ConGolog statements.

```
while ∃x.(hotel(x)∧ goodLoc(x)) do
        checkAvailability(x,dest,dDate,rDate)
endWhile

if ¬ hotelAvailable(dest,dDate,rDate) then
        BookB&B(cust,dest,dDate,rDate)
endIf

proc Travel(cust,origin,dest,dDate,rDate,purpose);
        If registrationRequired then Register;
        BookTransport(cust,origin,dest,dDate,rDate );
        BookAccommodations(cust,dest,dDate,rDate);
        UpdateExpenseClaim(cust);
        Inform(cust);
endProc
```

To instantiate a ConGolog program in the context of a *KB*, the abbreviation $Do(\delta,s,s')$ is defined. It says that $Do(\delta,s,s')$ holds whenever s' is a terminating situation following the execution of complex action δ, starting in situation s. Given a model, i.e. the agent KB, and a generic procedure δ, we can instantiate δ with respect to the KB and the current situation S_0 by entailing a binding for the situation variable s. Since situations are simply the history of actions from S_0, the binding for s defines a sequence of actions that leads to successful termination of the generic procedure δ.

$$KB \models (\exists s).Do(\delta, S_0, s)$$

Following [3], primitive actions, complex actions and generic procedures are first-order terms in our language. This is advantageous for constructing ontologies because we can talk and reason about these actions within the language.

It is important to observe that ConGolog programs, and hence our generic procedures, are not programs in the conventional sense. While they have the complex structure of programs, including loops, if-then-else statements etc., they differ in that they are not necessarily deterministic. Rather than necessarily dictating a unique sequence of actions, ConGolog programs serve to add temporal constraints to the situation tree of a KB, as depicted in Figure 3. As such, they eliminate certain branches of the situation tree (designated by the X's), reducing the size of the search space of situations that instantiate the generic procedure. The *Desirable* predicate, *Desirable(a,s)* [14] which we have introduced to incorporate user constraints also further reduces the tree to those situations that are desirable to the user.

Since generic procedures and customizing constraints simply serve to constrain the possible evolution of actions, then depending upon how they are specified, they can play different roles. At one extreme, the generic procedure simply constrains the search space required in planning. At the other extreme, a generic procedure can dictate a unique sequence of actions, much in the way a traditional program might. We leverage this nondeterminism to describe generic procedures that have the leeway to be relevant to a broad range of users, while at the same time being customizable to reflect the desires of individual users. We contrast this to a typical procedural program that would have to be explicitly modified to incorporate unanticipated constraints.

4.3 Implementation

To implement our agent technology, we started with an implementation of an online ConGolog interpreter in Quintus Prolog 3.2 [3]. First, we extended the interpreter to take personal constraints into consideration, i.e, the *Desirable* fluent. The interpreter was modified so that when constructing a plan, it would only consider actions (services) that were both possible and desirable. For example, if Bob asks his agent to arrange a trip out of town for the period December 1-3, but his personal schedule states that he must be home on December 2, *MustbeHome(December-2,s)*, the interpreter will respond to the attempted instantiation of the travel procedure with the answer 'No', since none of the possible plans satisfy the *MustbeHome* constraint. The interpreter is able to reason that the requested travel would entail Bob being away on December 2, which conflicts with his personal constraints.

We also modify the interpreter to create a middle ground between offline and online execution. As such it exploits both the merits of backtracking and the merits of online execution of information-gathering actions to reduce the search space size. Our interpreter performs information-gathering actions, such as `GetDrivingDistance` (Yahoo's™ driving direction service) or `GetFlightSchedule` (United Airlines'™ flight schedule information service) online to collect the relevant information needed in the ConGolog program, while only simulating the effects of world-altering actions such as `BuyUALTicket`. By only simulating rather than executing, the interpreter can backtrack if it finds that a decision it has made does not lead to successful termination of the program. (Humans often follow this approach when instantiating their generic procedures. They collect information (e.g., flight schedules, hotel availability) and choose a sequence of actions, relying on the information they have collected persisting until they actually execute the associated world-altering actions (e.g., booking the

flight or the hotel). If this persistence assumption fails, they replan.) Following successful generation of a terminating situation by our interpreter, the sequence of world-altering actions that comprise the terminating situation can be executed, and will result in successful execution of the ConGolog program, if our persistence assumptions have not been violated. Finally, we added new constructs to the ConGolog language to enable more flexible encoding of generic procedures. Details of these extensions can be found in [14].

The interpreter was also modified to communicate with the agent broker, OAA [12], to send requests for services to the appropriate Web services, and to dispatch responses to the agents. When the Semantic Web is a reality, Web services will communicate via DAML. Currently we must translate our markup (DAML+OIL and a subset of first-order logic) back and forth to HTML via a set of java programs. We use an information extraction program, World Wide Web Wrapper Factory (W4F)[3], to extract the information from the HTML currently generated by Web services. All information-gathering services are performed this way. For obvious practical and financial reasons, world-altering services are not actually executed.

Implementing a deductive reasoning system that operates on the semantic Web, for all practical purposes, an enormous KB that will no doubt be locally consistent but globally inconsistent, presents some interesting challenges. One challenge is with negation. Prolog implements negation-as-failure (NAF). E.g., if we can't prove there is a flight from San Francisco to Boston for under $300, then we infer that it is not the case that there is such a flight. Of course, on the Web, we cannot practically search the entire Web KB, so we must define a notion of local-closed-world NAF, which circumscribes our search space. If we cannot prove a proposition *p* in our locally defined KB (our context), then it is false. We have adopted a similar local-closed-world notion for knowledge. Following Kripke semantics, an agent is said to know something if it is true in all possible worlds the agent considers it could be in. Nevertheless, as with NAF, this becomes too difficult to manage, so instead we make a local-closed-world assumption, and infer that an agent knows a proposition *p* in situation *s* if it is true in the circumscribed set of worlds the agent considers it could be in.

4.4 Example

In this section we illustrate the execution of our agent technology with a generic procedure for making travel arrangements. Bob Chen wishes to travel from San Francisco to Monterey on Knowledge Systems Laboratory (KSL) business with the DAML project. He has two constraints – one personal and one inherited from the lab to which he belongs. He wishes to drive rather than fly, if the driving time is less than 3 hours, and as a member of the Knowledge Systems Laboratory (KSL), he has inherited the constraint that he must use an American carrier for business travel.

In reality, our demo doesn't provide much to see. The user makes a request to the agent through a user interface that is automatically created from our DAML+OIL agent procedures ontology, and the agent e-mails the user when it is done. For the purposes of illustration, Figure 4 provides a window into what is happening behind the scenes. It is a trace from the run of our augmented and extended ConGolog interpreter, operating in Quintus Prolog. The agent KB is represented in a Prolog encoding of the situation calculus, a translation of the semantic Web service markup relevant to the generic travel procedure being called, together with Bob's user constraint markup. We have defined a generic procedure for travel, not unlike the one illustrated in Section 4.2.

In our software demonstrations of this work, we show that the same generic procedure can be called by multiple users, each with different personal and group constraints. Depending upon these constraints, the agent's deductive machinery generates very different sequences of Web service calls that meet the individual needs of the user. This of course is due to the nondeterminacy in the generic procedure, and the way it is customized for individual users.

The first arrow points to the call of the ConGolog procedure `travel(user,origin,dest,dDate,rDate,purpose)`, with the parameters instantiated as noted. Arrow #2 shows the interpreter executing the information-gathering action `get_driving_time` by requesting the driving time from OAA, which sends a request to Yahoo Maps to execute its `GetDrivingTime(San Franciso, Monterey)` service. Yahoo Maps returns that the driving time between San Francisco and Monterey is 2 hours.

```
xterm
1 | ?- travel('Bob Chen','09/02/00','09/06/00','San Francisco','Monterey','DAML').
2 Contacting Web Service Broker:
      Request Driving Time [San Francisco] - [Monterey]
  Result 2

3 Contacting Web Service Broker:
       Request Car Info in [San Francisco]
  Result
     <B>HERTZ<B>Shuttle to Car Counter<B>Economy Car Automati...
     <B>ACE<B>Off Airport, Shuttle Provided<B>Economy Car Aut...
     <B>NATIONAL<B>Shuttle to Car Counter<B>Economy Car Autom...
     <B>FOX<B>Off Airport, Shuttle Provided<B>Mini Car Automa...
     <B>PAYLESS<B>Off Airport, Shuttle Provided<B>Mini Car Au...
     <B>ALL INTL<B>Off Airport, Shuttle Provided<B>Economy Ca...
     <B>HOLIDAY<B>Off Airport, Shuttle Provided<B>Economy Car...
     <B>ABLE RENT<B>Off Airport, Shuttle Provided<B>Compact C...

4 Select
       HERTZ (San Francisco Airport), Location: Shuttle to Car Counter, Economy C
  ar Automatic with Air Conditioning, Unlimited Mileage

5 Contacting Web Service Broker:
      Request Hotel Info in [Monterey]
  Result
     <B>Travelodge<B> Monterey, CA<B>65 Rooms / 2 Floors<B>No...
     <B>Econolodges<B>MONTEREY, CA<B>47 Rooms / 2 Floors<B> 1...
     <B>Lexington Services<B> Monterey, CA<B>52 Rooms<B>Not A...
     <B>Ramada Inns<B> Monterey, CA<B>47 Rooms<B>Not Availabl...
     <B>Best Western Intl<B> Monterey, CA<B>43 Rooms / 3 Floo...
     <B>Motel 6<B> Monterey, CA<B>52 Rooms / 2 Floors<B>Not A...
     <B>Villager Lodge<B> Monterey, CA<B>55 Rooms / 2 Floors<...
     <B>Best Western Intl<B>  Monterey, CA<B>34 Rooms / 2 Flo...
```

Figure 4. Agent Interacting with Web

Since Bob has a constraint that he wishes to drive if the driving distance is less than 3 hours, booking a flight is not *Desirable* and thus no selected by the interpreter. Consequently, as depicted at Arrow#3, the agent elects to search for an available car rental at the point of origin, San Francisco. It executes the information-gathering action `get_car_rental_info`, by requesting the car rental info from OAA, which sends a request to the appropriate Web service at www.travelocity.com. A number of available cars are returned, and since Bob has no constraints, the first car is selected at Arrow#4. Arrow #5 depicts the call to OAA for a hotel at the destination point, and so on.

Our agent technology goes on to complete Bob's travel arrangements, creating an expense claim form for Bob, filling in as much information as was available from the Web services. The expense claim illustrates the agent's ability to write semantic

[3] http://db.cis.upenn.edu/W4F

Web markup, in addition to reading it. Finally, the agent sends an e-mail message to Bob, notifying him of his agenda, and informing him that his expense claim has been created.

To appreciate one of the merits of our approach it is interesting to contrast this execution of the generic travel procedure with the exact same call by a different user, with different user constraints. Ruby Loo also wishes to travel from San Francisco to Monterey, but for personal reasons. She has the constraint that she likes to drive if the time is less than 2 hours. She also inherits KSL's constraint restricting air carriers for business travel. Using the same generic procedure, the agent executes a different sequence of Web services. After determining the driving time, the agent searches for flights at `www.travelocity.com`. Since there are no restrictions on the flights, it picks the first flight, then it books a car locally for Ruby in Monterey, and so on. Other permutations of business and personal travel by different users with different constraints yield a variety of different executions.

This concludes our discussion of our DAML-enabled agent technology for Web service composition. Once again, our objective was to describe our basic approach and the rationale behind it. Further details concerning this technology will appear in another publication (e.g., [14]) or at our Web site.

The agent technology presented here is broadly related to the plethora of work on agent-based systems and agent programming. Three agent technologies that deserve mention are the Golog family of agent technologies referenced above, the work of researchers at SRI on Web agent technology (e.g., [18]), and the softbot work developed at the University of Washington (e.g., [5]). The latter also used a notion of action schemas to describe information-gathering and world-altering actions on the Internet that an agent could use to plan to achieve a goal. Finally, also of note is the IBROW system, an intelligent brokering service for knowledge-component reuse on the World Wide Web (e.g., [1]). Our work is similar to IBROW in the use of an agent brokering system and ontologies to support interaction with the Web. Nevertheless, we differ in the details. We are focusing on developing and exploiting semantic Web markup, which will provide us with the KB for our agents. Our agent technology performs automated service composition based on this markup. This is not a problem the IBROW community is addressing at present.

5. SUMMARY

In this paper we presented a vision for the future of services on the semantic Web. We distinguished three automation tasks that will be enabled by the semantic markup of Web services: Web service discovery, Web service execution, and Web service composition and interoperation. Using these tasks as drivers, we presented an approach to semantic markup of Web services and user constraints in the DAML family of markup languages. This markup creates services that are computer-interpretable, use-apparent, and agent-ready. We also presented one agent technology that exploits this semantic markup to perform automatic Web service composition through the unique exploitation of generic procedures and customizing constraints.

6. ACKNOWLEDGEMENTS

The authors would like to thank Richard Fikes and Deborah McGuinness for useful discussions related to this work; Ron Fadel and Jessica Jenkins for their help with service ontology construction; and the reviewers, Adam Cheyer, and Karl Pfleger for helpful comments on a draft of this paper. We would also like to thank the Cognitive Robotics Group at the University of Toronto for providing an initial ConGolog interpreter that we have extended and augmented, and SRI for the use of the Open Agent Architecture (OAA) software. Finally we gratefully acknowledge the financial support of the US Defense Advanced Research Projects Agency DAML Program #F30602-00-2-0579-P00001.

7. REFERENCES

[1] V. R. Benjamins et al., "IBROW3 - An Intelligent Brokering Service for Knowledge-Component Reuse on the World Wide Web." *Proceedings of the Eleventh Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW'98),* Banff, Canada, 1998.

[2] T. Berners-Lee, M. Fischetti and T. M. Dertouzos, "Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its Inventor." Harper, San Francisco, September 1999.

[3] G. De Giacomo, Y. Lesperance and H. Levesque, "ConGolog, a concurrent programming language based on the situation calculus." *Artificial Intelligence*, 1-2 (121), pp. 109-169, 2000.

[4] G. Denker, J. R. Hobbs, D. Martin, S. Narayanan, and R. Waldinger, "Accessing Information and Services on the DAML-Enabled Web." *Proceedings of the Second International Workshop on the Semantic Web (SemWeb2001).* To appear, 2001.

[5] O. Etizioni and D. Weld, "A Softbot-based interface to the internet." *Communications of the ACM*, 72-76, July 1994.

[6] D. Fensel, "The Semantic Web and its Languages." *IEEE Intelligent Systems, Trends and Controversies*, pp. 1, November/December, 2000.

[7] M. Ghallab et al., "PDDL --- The Planning Domain Definition Language. Version 1.2." Yale Center for Computational Vision and Control, Tech Report CVC TR-98-003/DCS TR-1165, October, 1998.

[8] J. Hendler, "Agents on the Web." *IEEE Intelligent Systems* (forthcoming).

[9] J. Hendler and D. McGuinness, "The DARPA Agent Markup Language." *IEEE Intelligent Systems, Trends and Controversies*, pp. 6-7, November/December 2000.

[10] O. Lassila and R. Swick: "Resource Description Framework (RDF) Model and Syntax Specification", W3C Recommendation, World Wide Web Consortium, Cambridge (MA), February 1999; available on-line as http://www.w3.org/TR/REC-rdf-syntax/

[11] S. Luke and J. Heflin, "SHOE 1.01. Proposed Specification." http://www.cs.umd.edu/projects/plus/SHOE/spec1.01.html, February, 2000.

[12] D. L. Martin, A. J. Cheyer, and D. B. Moran, "The open agent architecture: A framework for building distributed software systems." *Applied Artificial Intelligence,* vol. 13, pp. 91--128, January-March 1999.

[13] S. McIlraith, "Modeling and Programming Devices and Web Agents." *Proceedings of the NASA Goddard Workshop on Formal Approaches to Agent-Based Systems*, Lecture Notes in Computer Science, Springer-Verlag. To appear, 2001.

[14] S. McIlraith and T. C. Son, "Programming the Semantic Web." Manuscript under review, 2001.

[15] R. Reiter. "KNOWLEDGE IN ACTION: Logical Foundations for Describing and Implementing Dynamical Systems." Book Draft, 2000; available online at http://www.cs.utoronto.ca/~cogrobo/

[16] K. Sycara, M. Klusch, S. Widoff and J. Lu, "Dynamic Service Matchmaking Among Agents in Open Information Environments." *Journal ACM SIGMOD Record (Special Issue on Semantic Interoperability in Global Information Systems)*, Vol. 28, No. 1, pp. 47-53, March 1999.

[17] F. van Harmelen and I. Horrocks, "FAQs on OIL: the Ontology Inference Layer." *IEEE Intelligent Systems, Trends and Controversies*, pp. 3-6, November/December 2000.

[18] R. Waldinger, "Deductive Composition of Web Software Agents." *Proceedings of the NASA Goddard Workshop on Formal Approaches to Agent-Based Systems*, Lecture Notes in Computer Science, Springer-Verlag. To appear, 2001.