Semantic Web Services

Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng, *Stanford University*

The authors propose the markup of Web services in the DAML family of Semantic Web markup languages. This markup enables a wide variety of agent technologies for automated Web service discovery, execution, composition, and interoperation. The authors present one such technology for automated Web service composition.

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 business-to-business applications. Web-accessible programs, databases, sensors, and a variety of other physical devices realize these services. In the next decade, computers will most likely be ubiquitous, and most devices will have some sort of computer inside them. Vint Cerf, one of the fathers of the Internet, views the population of the Internet by smart devices as the harbinger of a new revolution in Internet technology.

Today's Web was designed primarily for human interpretation and 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 having computer programs or agents[1] implement reliable, large-scale interoperation of Web services is the need to make such services computer interpretable—to create a Semantic Web[2] of services whose properties, capabilities, interfaces, and effects are encoded in an unambiguous, machine-understandable form.

The realization of the Semantic Web is underway with the development of new AI-inspired content markup languages, such as OIL,[3] DAML+OIL (www.daml.org/2000/10/daml-oil), and DAML-L (the last two are members of the DARPA Agent Markup Language (DAML) family of languages).[4] These languages have a well-defined semantics and enable the markup and manipulation of complex taxonomic and logical relations between entities on the Web. 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 article 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 family of Semantic Web markup languages.[4] Such so-called *semantic markup* of Web services creates a distributed knowledge base. This provides a means for agents to populate their local KBs so that they can reason about Web services to perform automatic Web service discovery, execution, and composition and interoperation.

To illustrate this claim, we present an agent technology based on reusable generic procedures and customizing user constraints that exploits and showcases our Web service markup. This agent technology is realized using the first-order language of the situation calculus and an extended version of the agent programming language ConGolog,[5] together with deductive machinery.

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 generic procedures. In addition to the markup, our framework includes a variety of agent tech-

1094-7167/01/$10.00 © 2001 IEEE

nologies—specialized services that use an agent broker to send requests for service to appropriate Web services and to dispatch service responses back to the agent.

Automating Web services

To realize our vision of Semantic Web services, we are creating semantic markup of Web services that makes them machine understandable and use-apparent. We are also developing agent technology that exploits this semantic markup to support automated Web service composition and interoperability. Driving the development of our markup and agent technology are the automation tasks that semantic markup of Web services will enable—in particular, service discovery, execution, and composition and interoperation.

Automatic Web service discovery involves automatically locating Web services that provide a particular service and that adhere to requested properties. A user might say, for example, "Find a service that sells airline tickets between San Francisco and Toronto and that accepts payment by Diner's Club credit card." Currently, a human must perform this task, first using a search engine to find a service and then either reading the Web page associated with that service or executing the service to see whether it adheres to the requested properties. With semantic markup of services, we can specify the information necessary for Web service discovery as computer-interpretable semantic markup at the service Web sites, and a service registry or (ontology-enhanced) search engine can automatically locate appropriate services.

Automatic Web service execution involves a computer program or agent automatically executing an identified Web service. A user could request, "Buy me an airline ticket from www.acmetravel.com on UAL Flight 1234 from San Francisco to Toronto on 3 March." To execute a particular service on today's Web, such as buying an airline ticket, a user generally must go to the Web site offering that service, fill out a form, and click a button to execute the service. Alternately, the user might send an http request directly to the service URL with the appropriate parameters encoded. Either case requires a human to understand what information is required to execute the service and to interpret the information the service returns. Semantic markup of Web services provides a declarative, computer-interpretable API for executing services. The markup tells the agent what input is necessary, what information will be returned, and how to execute—and potentially interact with—the service automatically.

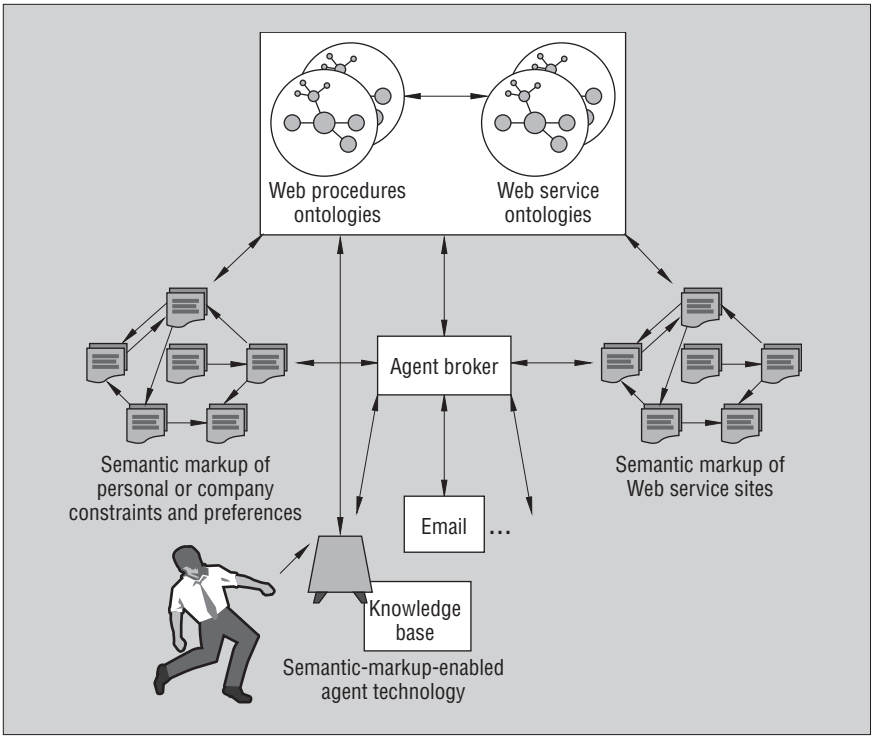

Figure 1. A framework for Semantic Web services.

Automatic Web service composition and interoperation involves the automatic selection, composition, and interoperation of appropriate Web services to perform some task, given a high-level description of the task's objective. A user might say, "Make the travel arrangements for my IJCAI 2001 conference trip." 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 (for example, selecting a flight from among several options). With semantic markup of Web services, the information necessary to select, compose, and respond to services is encoded at the service Web sites. We can write software to manipulate this markup, together with a specification of the task's objectives, to achieve the task automatically. Service composition and interoperation leverage automatic discovery and execution.

Of these three tasks, none is entirely realizable with today's Web, primarily because of a lack of content markup and a suitable markup language. Academic research on Web service discovery is growing out of agent matchmaking research such as the Lark system,[6] which proposes a representation for annotating agent capabilities so that they can be located and brokered. Recent industrial efforts have focused 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 and OASIS (Organization for the Advancement of Structured Information Standards) to standardize a framework for trading partner interchange.

E-business infrastructure companies are 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. 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 its OSM Platform—is 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. Our semantic

markup 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 or application level. To demonstrate this claim, we present agent technology that performs automatic Web service composition, an area that industry is not yet tackling in any great measure.

Semantic Web service markup

The three automation tasks we've described are driving the development of our semantic Web services markup in the DAML family of markup languages. We are marking up

- Web services, such as Yahoo's driving direction information service or United Airlines' flight booking service;
- user and group constraints and preferences, such as a user's—let's say Bob's—schedule, that he prefers driving over flying if the driving time to his destination is less than three hours, his use of stock quotes exclusively from the E*Trade Web service, and so forth; and
- agent procedures, which are (partial) compositions of existing Web services, designed to perform a particular task and marked up for sharing and reuse by groups of other users. Examples include Bob's business travel booking procedure or his friend's stock assessment procedure.

Our DAML markup provides a declarative representation of Web service and user constraint knowledge. (See the "The Case for DAML" sidebar to learn why we chose the DAML family of markup languages.) A key feature of our markup is the exploitation of ontologies, which DAML+OIL's roots in description logics and frame systems support.

We use ontologies to encode the classes and subclasses of concepts and relations pertaining to services and user constraints. (For example, the service BuyUALTicket and BuyLufthansaTicket are subclasses of the service BuyAirlineTicket, inheriting the parameters customer, origin, destination, and so forth). Domain-independent Web service ontologies are augmented by domain-specific ontologies that inherit concepts from the domain-independent ontologies and that additionally encode concepts that are specific to the individual Web service or user. Using 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- or user-specific markup by enabling an individual service or user to inherit much of its semantic markup from ontologies, thus requiring only minimal markup at the Web site. Most importantly, ontologies can give semantics to markup by constraining or grounding its interpretation. Web services and users need not exploit Web service ontologies, but we foresee many domains where communities will want to agree on a standard definition of terminology and encode it in an ontology.

DAML markup of Web services

Collectively, our markup of Web services provides

- declarative advertisements of service properties and capabilities, which can be used for automatic service discovery;
- declarative APIs for individual services that are necessary for automatic 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 KB of Web services. Semantic markup can populate detailed registries of the properties and capabilities of Web services for knowledge-based indexing and retrieval of Web services by agent brokers and humans alike. Semantic markup can also populate individual agent KBs, to enable automated reasoning about Web services.

Our Web service markup comprises a number of different ontologies that provide the backbone for our Web service descriptions. We define the domain-independent 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, or device that does not call another Web service. There is no ongoing interaction between the user and a primitive service. The user or agent calls the service, and the service returns a response. 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 services, so that the user can make decisions. An example might be interacting with www.amazon.com to buy a book.

Domain-specific Web service ontologies are subclasses of these general classes. They enable an individual service to inherit shared concepts, and vocabulary in a particular domain. The ontology being used is specified in the Web site markup and then simply refined and augmented to provide service-specific markup. For example, we might define an ontology containing the class Buy, with subclass BuyTicket, which has subclasses BuyMovieTicket, BuyAirlineTicket, and so forth. BuyAirlineTicket 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 the parameter Customer. BuyAirlineTicket will inherit the Customer parameter and will also have the parameters Origin, Destination, DepartureDate, and so on. We constructed domain-specific ontologies to describe parameter values. For example, we restricted the values of Origin and Destination 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 and simple type checking for our declarative APIs. In addition, we have used these restrictions in our agent technology to create customized user interfaces.

Markup for Web service discovery. To automate Web service discovery, we associate properties with services that are relevant to automated service classification and selection. In the case of BuyUALTicket, these would include service-independent property types such as the company name, the service URL, a unique service identifier, the intended use, and so forth. They would also include service-specific property types such as valid methods of payment, travel bonus plans accepted, and so forth. This markup, together with certain of the properties specified later, collectively provides a declarative advertisement of service properties and capabilities, which is computer interpretable and can be used for automatic service discovery.

Markup for Web service execution. To automate Web service execution, markup must

The Case for DAML

In recent years, several 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.[1]

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 might be different from the <DUE-DATE> at the purchaser's site.

The World Wide Web Consortium developed the resource description framework (RDF)[2] as a standard for metadata. The goal was to add a formal semantics to the Web, 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 nameable by a URI) in terms of named properties and values. The RDF working group also developed RDF Schema, 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, you can't define properties of properties, necessary and sufficient conditions for class membership, or equivalence and disjointness of classes. Furthermore, the only constraints expressible are domain and range constraints on properties. Finally, and perhaps most importantly, the semantics remains underspecified.

Recently, there have been several efforts to build on RDF and RDFS with more AI-inspired knowledge representation languages such as SHOE,[3] DAML-ONT,[4] OIL,[5] and most recently DAML+OIL. DAML+OIL is the second in the DAML family of markup languages, replacing DAML-ONT as an expressive ontology description language for markup. Building on top of RDF and RDFS, and with its roots in AI description logics, DAML+OIL overcomes many of the expressiveness inadequacies plaguing RDFS and most important, has a well-defined model-theoretic semantics as well as an axiomatic specification that determines the language's intended interpretations. 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 six months, 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 a flight information service that encodes whether a flight shows a movie. One way to do this is to create a markup for each flight indicating whether or not it does. A more compact representation is to write the constraint flight-over-3-hours $\rightarrow$ movie and to use deductive reasoning to infer if a flight will show a movie. This representation is more compact, more informative, and easier to modify than an explicit enumeration of individual flights and movies. Similarly, such clauses can represent markup constraints, business rules, and user preferences 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 lets a more expressive logical language extend DAML-L or act as an alternate extension to DAML+OIL. Because DAML-L has not yet been developed, our current Web service markup is in a combination of DAML+OIL and a subset of first-order logic. Our markup will evolve as the DAML family of languages evolves.

References

1. D. Fensel, "The Semantic Web and Its Languages," *IEEE Intelligent Systems*, vol. 15, no. 6, Nov./Dec. 2000, p. 67–73.
2. O. Lassila and R. Swick, *Resource Description Framework (RDF) Model and Syntax Specification*, W3C Recommendation, World Wide Web Consortium, Feb. 1999; www.w3.org/TR/REC-rdf-syntax (current 11 Apr. 2001).
3. S. Luke and J. Heflin, *SHOE 1.01. Proposed Specification*, www.cs.umd.edu/projects/plus/SHOE/spec1.01.html, 2000 (current 20 Mar. 2001).
4. J. Hendler and D. McGuinness, "The DARPA Agent Markup Language," *IEEE Intelligent Systems*, vol. 15, no. 6, Nov./Dec. 2000, pp. 72–73.
5. F. van Harmelen and I. Horrocks, "FAQs on OIL: The Ontology Inference Layer," *IEEE Intelligent Systems*, vol. 15, no. 6, Nov./Dec. 2000, pp. 69–72.

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. For example, if the user orders a book, the response will differ depending on 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 might require an exchange of information between the agent and an individual service. For example, a complex service that books a flight for a user might involve first finding flights that meet the user's request, then suspending until the user selects one flight. Complex services are composed of primitive or complex services using typical programming languages and business-process modeling language constructs such as Sequence, Iteration, If-then-Else, and so forth. 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. For an agent to respond automatically to a complex service execution—that is, to automatically interoperate with that service—it will require some of the information encoded for automatic composition and interoperation.

Markup for Web service composition. The function metaphor used for automatic Web service execution provides information about data flow, but it does not provide information about what the Web service actually does. To automate service composition, and for services and agents to interoperate, we must also encode how the service affects the world. For example, when a user visits www.amazon.com and

successfully executes the BuyBook service, she knows she has purchased a book, that her credit card will be debited, and that she will receive a book at the address she provided. Such consequences of Web service execution are not part of the markup nor part of the function-based specification provided for automatic execution. To automate Web service composition and interoperation, or even to select an individual service to meet some objective, we must encode prerequisites and consequences of Web service execution for computer use.

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

An advantage of exploiting an action metaphor to describe Web services is that it lets us bring to bear the vast AI research on reasoning about action, to support automated reasoning tasks such as Web service composition. 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.

Complex actions, like complex services, are compositions of individual services; however, dependencies between these compositions are predicated on state rather than on data, as is the case with the execution-motivated markup. Complex actions are composed of primitive actions or other complex actions using typical programming languages and business-process modeling-language constructs such as Sequence, Parallel, If-then-Else, While, and so forth.

DAML markup of user constraints and preferences

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 three hours, that he likes to get stock quotes from the E*Trade Web service, that his company requires all domestic business travel to be with a particular set of carriers, and so forth. The actual markup of user constraints is relatively straightforward, given DAML-L. We can express most constraints as these Horn clauses (see the sidebar), and ontologies let users classify, inherit, and share constraints. Inheriting terminology from Web service 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. More challenging than the markup itself is the agent technology that will appropriately exploit it.

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.

DAML-enabled agent technology

Our semantic markup of Web services enables a wide variety of agent technologies. Here, we present an agent technology we are developing that exploits DAML markup of Web services to perform automated Web service composition.

Consider the example task given earlier: "Make the travel arrangements for my IJCAI 2001 conference trip." If you were to perform this task using services available on the Web, you might first find the IJCAI 2001 conference Web page and determine the conference's location and dates. Based on the location, you would choose 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 so on.

Although the entire procedure is lengthy and somewhat tedious to perform, the average person could easily describe how to make your travel arrangements. Nevertheless, it's not easy to get someone else to make the arrangements for you. What makes this task difficult is not the basic steps but the need to make decisions to customize the generic procedure to enforce the traveler's constraints. Constraints can be numerous and consequently difficult for another person to keep in mind and satisfy. Fortunately, enforcing 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 of user constraints and preferences. We argue that many of the activities users might wish to perform on the Semantic Web, within the context of their workplace or home, can be viewed as customizations of reusable, high-level generic procedures. Our vision is to construct such reusable, high-level *generic procedures* and to represent them as distinguished services in DAML using a subset of the markup designed for complex services. We also hope to archive them in sharable generic procedures ontologies so that multiple users can access them. Generic procedures are customized with respect to users' constraints, using deductive machinery.

Generic procedures and customizing user constraints

We built our research on model-based programming[8] and on research into the agent programming language Golog and its variants, such as ConGolog.[5] Our goal was to provide a DAML-enabled agent programming capability that supports writing generic procedures for Web service-based tasks.

Model-based programs comprise a *model*—in this case, the agent's KB—and a *program*—the generic procedure we wish to execute. We argue that the situation calculus (a logical language for reasoning about action and change)

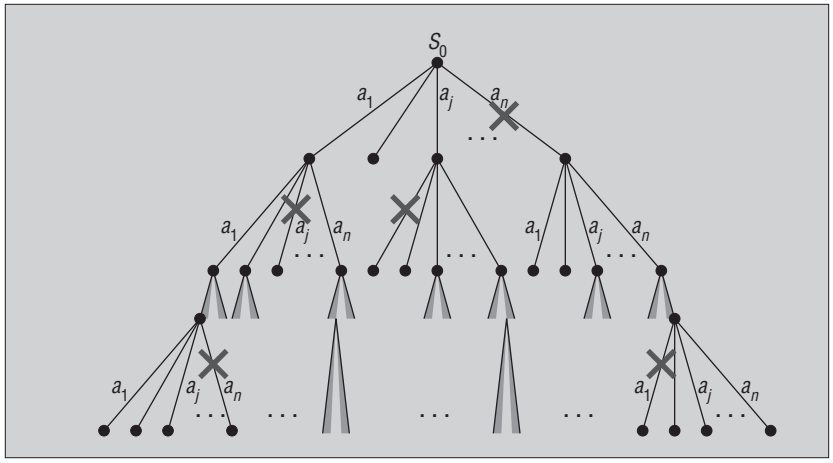

Figure 2. The tree of situations.

```
Primitive action: a
Test of truth: φ?
Sequence: (δ1; δ2)
Nondeterministic choice between actions: (δ1 | δ2)
Nondeterministic choice of arguments: π x.δ
Nondeterministic iteration: δ*
Conditiona: if φ then δ1 else δ2 endIf
Loop: while φ do δ endWhile
Procedure: proc P(v) δ endProc
```

(a)

```
while ∃x.(hotel(x)∧ goodLoc(x,dest)) do
        checkAvailability(x,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 endIf;
        BookTranspo(cust,origin,dest,dDate,rDate);
        BookAccommodations(cust,dest,dDate,rDate);
        UpdateExpenseClaim(cust);
        Inform(cust)
endProc
```

(b)

Figure 3. (a) A subset of the constructs in the ConGolog language. (b) Examples of ConGolog statements.

and ConGolog[5] provide a compelling 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 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 achieve but not exactly how to do it. They use procedural programming language constructs (**if-then-else**, **while**, and so forth) 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. We must deductively instantiate it in the context of the agent's KB, which includes properties of the agent and its user, properties of the specific services we are using, and the state of the world. We perform the instantiation by using deductive machinery. An instantiated program is simply a sequence of primitive actions (individual Web services), which ConGolog interprets and sends to the agent broker as a request for service. The great advantage of these generic procedures is that the same generic procedure, called with different parameters and user constraints, can generate very different sequences of actions.

ConGolog

ConGolog is a high-level logic programming language developed at the University of Toronto. Its primary use is for robot programming and to support high-level robot task planning. ConGolog is built on top of situation calculus. In situation calculus, 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 (for example, Web services such as **BuyUALTicket(origin,dest, date)**). Thus, a situation s is a history of the actions performed from S_0. The state of the world is expressed in terms of relations and functions (so-called *fluents*) that are true or false or have a particular value in a situation, s (for example, **flightAvailable(origin,dest, date,s)**).

Figure 2 illustrates the tree of situations induced by a situation calculus theory with actions $a_1, \ldots, a_n$ (ignore the ×'s for the time being). The tree is not actually computed, but it reflects the search space the situation calculus KB induces. We could have performed deductive plan synthesis to plan sequences of Web service actions over this search space, but instead, we developed generic procedures in ConGolog.

ConGolog provides a set of extralogical procedural programming constructs for assembling primitive and complex situation calculus actions into other complex actions.[5] Let δ_1 and δ_2 be complex actions, and let φ and a be so-called *pseudo fluents* and *pseudo actions*, respectively—that is, a fluent or action in the language of situation calculus with all its situation arguments suppressed. Figure 3a shows a subset of the constructs in the ConGolog language.

A user can employ these constructs to write generic procedures, which are complex actions in ConGolog. The instruction set for these complex actions is simply the general Web services (for example, **BookAirlineTicket**) or other complex actions. Figure 3b gives examples of ConGolog statements.

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 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. Because 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:

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

It is important to observe that ConGolog programs—and hence our generic procedures—are not programs in the conventional sense. Although they have the complex structure of programs—including loops, if-then-else statements, and so forth—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 Figure 2 depicts. As such, they eliminate certain branches of the situation tree (designated by the ×'s), reducing the size of the search space of situations that instantiate the generic procedure.

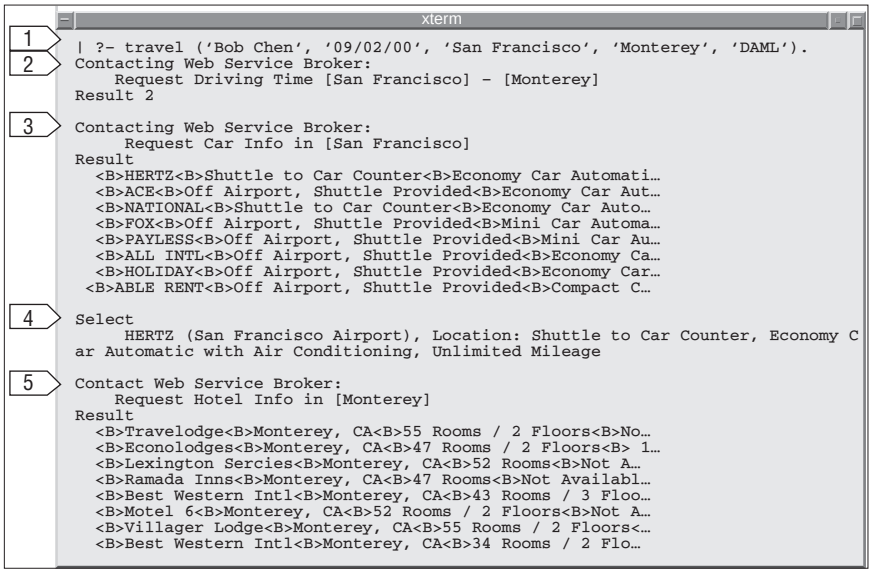

Figure 4. Agent interacting with Web services through OAA.

The *Desirable* predicate, *Desirable(a,s)*, which we introduced into ConGolog to incorporate user constraints, also further reduces the tree to those situations that are desirable to the user. Because generic procedures and customizing user constraints simply serve to constrain the possible evolution of actions, depending on 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 traditional procedural program that would have to be explicitly modified to incorporate unanticipated constraints.

Implementation

To implement our agent technology, we started with an implementation of an online ConGolog interpreter in Quintus Prolog 3.2.[5] We augmented and extended this interpreter in a variety of ways (discussed further elsewhere[9]). Some of the issues we dealt with were balancing the offline search for an instantiation of a generic procedure with online execution of information-gathering Web services, because they help to further constrain the search space of possible solutions. We added new constructs to the ConGolog language to enable more flexible encoding of generic procedures, and we incorporated users' customizing constraints into ConGolog by adding the *Desirable* predicate mentioned earlier.

We also modified the interpreter to communicate with the Open Agent Architecture agent brokering system.[10] OAA sends requests to appropriate Web services and dispatches responses to the agents. When the Semantic Web is a reality, Web services will communicate through DAML. Currently, we must translate our markup (DAML+OIL and a subset of first-order logic) back and forth to HTML through a set of Java programs. We use an information extraction program, World Wide Web Wrapper Factory (http://db.cis.upenn.edu/W4F), to extract the information Web services currently produce in HTML. All information-gathering services are performed this way. For obvious practical and financial reasons, world-altering aspects of services are not actually executed.

Example

Here, we illustrate the execution of our agent technology with a generic procedure for making travel arrangements. Let's say Bob wants to travel from San Francisco to Monterey on Knowledge Systems Lab business with the DARPA-funded DAML research project. He has two constraints—one personal and one inherited from the KSL, to which he belongs. He wishes to drive rather than fly, if the driving time is less than three hours, and as a member of the 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 emails the user the travel itinerary 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 Figure 3b.

Arrow 1 points to the call to the ConGolog procedure **travel(user,origin,dest,dDate,rDate,purpose)**, with the parameters instantiated as noted. Arrow 2 shows the interpreter contacting OAA, which sends a request to Yahoo Maps to execute the **getDrivingTime(San Franciso,Monterey)** service Yahoo Maps provides. Yahoo Maps indicates that the driving time between San Francisco and Monterey is two hours. Because Bob has a constraint that he wishes to drive if the driving distance is less than three hours, booking a flight is not desirable. Consequently, as depicted at Arrow 3, the agent elects to search for an available car rental at the point of origin, San Francisco. A number of available cars are returned, and because Bob has no constraints that affect car selection, 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 and filling in as much information as was available from the Web services. The expense claim illustrates the agent's ability to both read and write Semantic Web markup. Finally, the agent sends an email message to Bob, notifying him of his agenda.

To demonstrate the merits of our approach, we often contrast such an execution of the generic travel procedure with one a different user called, with different user constraints. The different user and constraints produce a different search space, thus yielding a different sequence of Web services.

Related work

Our agent technology broadly relates to the plethora of work on agent-based systems. Three agent technologies that deserve mention are the Golog family of agent technologies referenced earlier, the work of researchers at SRI on Web agent technology,[11] and the softbot work developed at the University of Washington.[12] The last also used a notion of action schemas to describe actions on the Internet that an agent could use to achieve a

goal. Also of note is the Ibrow system, an intelligent brokering service for knowledge-component reuse on the Web.[13] Our work is similar to Ibrow in the use of an agent brokering system and ontologies to support interaction with the Web. Nevertheless, 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 a problem the Ibrow community has yet to address.

The DAML family of semantic Web markup languages will enable Web service providers to develop semantically grounded, rich representations of Web services that a variety of different agent architectures and technologies can exploit to a variety of different ends. The markup and agent technology presented in this article is but one of many possible realizations. We are building on the markup presented here to provide a core set of Web service markup language constructs in a language we're calling DAML-S. We're working in collaboration with SRI, Carnegie Mellon University, Bolt Baranek and Newman, and Nokia, and we'll eventually publish the language at www.daml.org. Our agent technology for automating Web service composition and interoperation is also fast evolving. We'll publicize updates at www.ksl.stanford.edu/projects/DAML/webservices.

Acknowledgments

We 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 article. We also 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 software. Finally, we gratefully acknowledge the financial support of the US Defense Advanced Research Projects Agency DAML Program #F30602-00-2-0579-P00001.

References

1. J. Hendler, "Agents and the Semantic Web," *IEEE Intelligent Systems*, vol. 16, no. 2, Mar./Apr. 2001, pp. 30–37.
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, 1999.
3. F. van Harmelen and I. Horrocks, "FAQs on OIL: The Ontology Inference Layer," *IEEE Intelligent Systems*, vol. 15, no. 6, Nov./Dec. 2000, pp. 69–72.
4. J. Hendler and D. McGuinness, "The DARPA Agent Markup Language," *IEEE Intelligent Systems*, vol. 15, no. 6, Nov./Dec. 2000, pp. 72–73.
5. G. De Giacomo, Y. Lesperance, and H. Levesque, "ConGolog, a Concurrent Programming Language Based on the Situation Calculus," *Artificial Intelligence*, vols. 1–2, no. 121, Aug. 2000, pp. 109–169.
6. K. Sycara et al., "Dynamic Service Matchmaking among Agents in Open Information Environments," *J. ACM SIGMOD Record*, vol. 28, no. 1, Mar. 1999, pp. 47–53.
7. M. Ghallab et al., *PDDL: The Planning Domain Definition Language, Version 1.2*, tech. report CVC TR–98–003/DCS TR–1165, Yale Center for Computational Vision and Control, Yale Univ., New Haven, Conn., 1998.
8. S. McIlraith, "Modeling and Programming Devices and Web Agents," to be published in *Proc. NASA Goddard Workshop Formal Approaches to Agent-Based Systems, Lecture Notes in Computer Science*, Springer-Verlag, New York, 2001.
9. S. McIlraith and T.C. Son, "Adapting Golog for Programming the Semantic Web," to be published in *Proc. 5th Symp. on Logical Formalizations of Commonsense Reasoning* (Common Sense 2001), 2001.
10. 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, nos. 1–2, Jan.–Mar. 1999, pp. 91–128.
11. R. Waldinger, "Deductive Composition of Web Software Agents," to be published in *Proc. NASA Goddard Workshop Formal Approaches to Agent-Based Systems, Lecture Notes in Computer Science*, Springer-Verlag, New York, 2001.
12. O. Etzioni and D. Weld, "A Softbot-Based Interface to the Internet," *Comm. ACM*, July 1994, Vol. 37, no. 7, pp. 72–76.
13. V. R. Benjamins et al., "IBROW3: An Intelligent Brokering Service for Knowledge-Component Reuse on the World Wide Web," *Proc. 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop* (KAW '98), Banff, Canada, 1998; http://spuds.cpsc.ucalgary.ca/KAW/KAW98/KAW98Proc.html (current 20 Mar. 2001).

The Authors

Sheila A. McIlraith is a research scientist in Stanford University's Knowledge Systems Laboratory and the project lead on the KSL's DAML Web Services project. Her research interests include knowledge representation and reasoning techniques for the Web, for modeling, diagnosing, and controlling static and dynamical systems, and for model-based programming of devices and agents. She received her PhD in computer science from the University of Toronto. Contact her at sam@ksl.stanford.edu.

Tran Cao Son is an assistant professor in the Department of Computer Science at New Mexico State University. His research interests include knowledge representation, autonomous agents, reasoning about actions and changes, answer set programming and its applications in planning and diagnosis, model based reasoning, and logic programming. Contact him at tson@cs.nmsu.edu.

Honglei Zeng is a graduate student in the Department of Computer Science at Stanford University. He is also a research assistant in the Knowledge Systems Laboratory. His research interests include the Semantic Web, knowledge representation, commonsense reasoning, and multiple agents systems. Contact him at hlzeng@ksl.stanford.edu.