The Semantic Web: A Brain for Humankind

Dieter Fensel, *Vrije Universiteit Amsterdam*
Mark A. Musen, *Stanford University*

Originally, the computer was intended as a device for computation. Then, in the 1980s, the PC developed into a system for games, text processing, and PowerPoint presentations. Eventually, the computer became a portal to cyberspace—an entry point to a worldwide network of information exchange and business transactions. Consequently, technology that supports access to unstructured, heterogeneous, and distributed information and knowledge sources is about to become as essential as programming languages were in the 60s and 70s.

The Internet—especially World Wide Web technology—was what introduced this change. The Web is an impressive success story, in terms of both its available information and the growth rate of human users. It now penetrates most areas of our lives, and its success is based on its simplicity. The restrictiveness of HTTP and (early) HTML gave software developers, information providers, and users easy access to new media, helping this media reach a critical mass.

Unfortunately, this simplicity could hamper further Web development. What we're seeing is just the first version of the Web. The next version will be even bigger and more powerful—but we're still figuring out how to obtain this upgrade.

Growing complexity

Figure 1 illustrates the growth rate of current Web technology. It started as an in-house solution for a small group of users. Soon, it established itself as a worldwide communication medium for more than 10 million people. In a few years, it will interweave one billion people and penetrate not just computers but also other devices, including cars, refrigerators, coffee machines, and even clothes.

However, the current state of Web technology generates serious obstacles to its further growth. The technology's simplicity has already caused bottlenecks that hinder searching, extracting, maintaining, and generating information. Computers are only used as devices that post and render information—they don't have access to the actual content. Thus, they can only offer limited support in accessing and processing this information. So, the main burden not only of accessing and processing information but also of extracting and interpreting it is on the human user.

The Semantic Web

Tim Berners-Lee first envisioned a Semantic Web that provides automated information access based on machine-processable semantics of data and heuristics that use these metadata. The explicit representation of the semantics of data, accompanied with domain theories (that is, *ontologies*), will enable a Web that provides a qualitatively new level of service. It will weave

1094-7167/01/$10.00 © 2001 IEEE

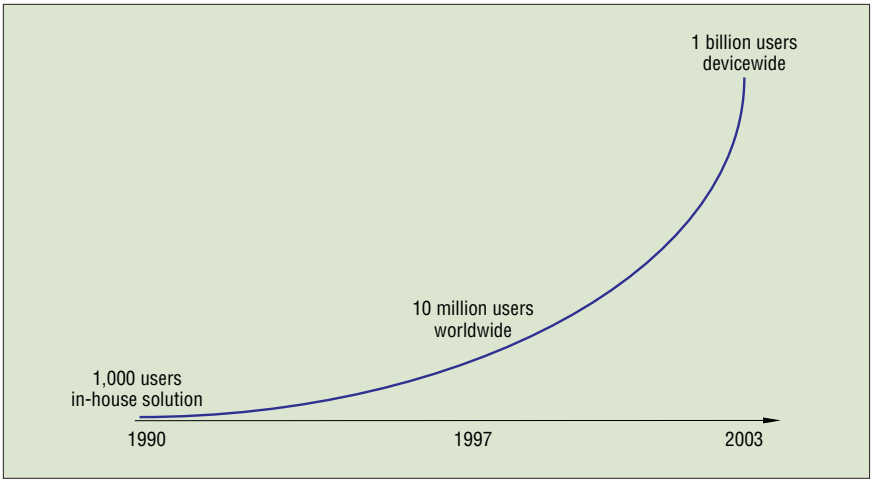


Figure 1. The growth rate of current Web technology.

together an incredibly large network of human knowledge and will complement it with machine processability. Various automated services will help the user to achieve goals by accessing and providing information in a machine-understandable form. This process might ultimately create an extremely knowledgeable system with various specialized reasoning services—systems that can support us in nearly all aspects of our life and that will become as necessary to us as access to electric power.

This gives us a completely new perspective of the knowledge acquisition and engineering and the knowledge representation communities. Some 20 years ago, AI researchers coined the slogan "knowledge is power." Quickly, two communities arose:

- knowledge acquisition and engineering, which deals with the bottleneck of acquiring and modeling knowledge (the human-oriented problem), and
- knowledge representation, which deals with the bottleneck of representing knowledge and reasoning about it (the computer-oriented problem).

However, the results of both communities never really hit the nail on the head. Knowledge acquisition is too costly, and the knowledge representation systems that were created were mainly isolated, brittle, and small solutions for minor problems.

With the Web and the Semantic Web, this situation has changed drastically. We have millions of knowledge "acquisitioners" working nearly for free, providing up to a billion Web pages of information and knowledge. Transforming the Web into a "knowledge Web" suddenly put knowledge acquisition and knowledge representation at the center of an extremely interesting and powerful topic: Given the amount of available online information we already have achieved, this Knowledge (or Semantic) Web will be extremely useful and powerful. Imagine a Web that contains large bodies of the overall human knowledge and trillions of specialized reasoning services using these bodies of knowledge. Compared to the potential of the Semantic Web, the original AI vision seems small and old-fashioned, like an idea of the 19th century. Instead of trying to rebuild some aspects of a human brain, we are going to build a brain of and for humankind.

In this issue

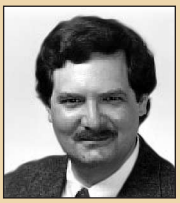
The work and projects described in this special issue provide initial steps into such a direction. We start with Michel Klein's tutorial, which introduces the current language standards of the Semantic Web: XML, XMLS, RDF, and RDFS.

James Hendler—who has already helped us all by successfully initiating and running a large DARPA-funded initiative on the Semantic Web—reveals his vision of the Semantic Web. On the basis of a standard ontology language, he sees software agents populating the Semantic Web, providing intelligent services to their human users. In "OIL: An Ontology Infrastructure for the Semantic Web," Dieter Fensel, Ian Horrocks, Frank van Harmelen, Deborah L. McGuinness, and Peter F. Patel-Schneider propose such an ontology standard language. OIL and DAML+OIL are the basis of a semantic working group of the W3C that should soon develop a standardization approach. Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng, in "Semantic Web Services," and Jeff Heflin and James Hendler, in "A Portrait of the Semantic Web in Action," describe intelligent services on top of such services, based on query and reasoning support for the Semantic Web.

A key technology for the Semantic Web is ontologies. In "Creating Semantic Web Contents with Protégé-2000," Natalya F. Noy, Michael Sintek, Stefan Decker, Monica Crubézy, Ray W. Fergerson, and Mark A. Musen provide excellent tool support for manually building ontologies based on Protégé-2000. However, even with an excellent tool environment, manually building ontologies is labor intensive and costly. Alexander Maedche and Steffen Staab, in "Ontology Learning for the Semantic Web," try to mechanize The ontology building process with machine learning techniques. ■

The Authors

Dieter Fensel is an associate professor at the Division of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, and a new department editor for Trends & Controversies. After studying mathematics, sociology, and computer science in Berlin, he joined the Institute AIFB at the University of Karlsruhe. His major subject was knowledge engineering, and his PhD thesis was on formal specification language for knowledge-based systems. Currently, his focus is on using ontologies to mediate access to heterogeneous knowledge sources and to apply them in knowledge management and e-commerce. Contact him at the Division of Mathematics and Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, Netherlands; dieter@cs.vu.nl; www.cs.vu.nl/~dieter.



Mark A. Musen is an associate professor of medicine (medical informatics) and computer science at Stanford University and is head of the Stanford Medical Informatics laboratory. He conducts research related to knowledge acquisition for intelligent systems, knowledge-system architecture, and medical-decision support. He has directed the Protégé project since its inception in 1986, emphasizing the use of explicit ontologies and reusable problem-solving methods to build robust knowledge-based systems. He has an MD from Brown University and a PhD from Stanford. Contact him at Stanford Medical Informatics, 251 Campus Dr., Stanford Univ., Stanford, CA 94305; musen@smi.stanford.edu; www.smi.stanford.edu/people/musen.

By Michel Klein
Vrije Universiteit

XML, RDF, and Relatives

Languages for representing data and knowledge are an important aspect of the Semantic Web. And there are a lot of languages around! Most languages are based on XML or use XML as syntax; some have connections to RDF or RDF Schemas. This tutorial will briefly introduce XML, XML Schemas, RDF, and RDF Schemas.

Let's start with XML

XML (eXtensible Markup Language) is a specification for computer-readable documents. *Markup* means that certain sequences of characters in the document contain information indicating the role of the document's content. The markup describes the document's data layout and logical structure and makes the information self-describing, in a sense. It takes the form of words between pointy brackets, called *tags*—for example, <name> or <h1>. In this aspect, XML looks very much like the well-known language HTML.

However, *extensible* indicates an important difference and a main characteristic of XML. XML is actually a *metalanguage*: a mechanism for representing other languages in a standardized way. In other words, XML only provides a data format for structured documents, without specifying an actual vocabulary. This makes XML universally applicable: you can define customized markup languages for unlimited types of documents. This has already occurred on a massive scale. Besides many proprietary languages—ranging from electronic order forms to application file formats—a number of standard languages are defined in XML (called *XML applications*). For example, XHTML is a redefinition of HTML 4.0 in XML.

Let's take a more detailed look at XML. The main markup entities in XML are *elements*. They consist normally of an opening tag and a closing tag—for example, <person> and </person>. Elements might contain other elements or text. If an element has no content, it can be abbreviated as <person/>. Elements should be properly nested: a child element's opening and closing tags must be within its parent's opening and closing tags. Every XML document must have exactly one root element. Elements can carry *attributes* with values, encoded as additional "word = value" pairs inside an element tag—for example, <person name="John">. Here is a piece of XML:

```
<?xml version="1.0"?>
<employees>
    List of persons in company:
    <person name="John">
        <phone>47782</phone>
        On leave for 2001.
    </person>
</employees>
```

XML does not imply a specific interpretation of the data. Of course, on account of the tag's names, the meaning of the previous piece of XML seems obvious to human users, but it is not formally specified! The only legitimate interpretation is that XML code contains named entities with subentities and values; that is, every XML document forms an ordered, labeled tree. This generality is both XML's strength and its weakness. You can encode all kinds of data structures in an unambiguous syntax, but XML does not specify the data's use and semantics. The parties that use XML for their data exchange must agree beforehand on the vocabulary, its use, and its meaning.

Enter DTDs and XML Schemas

Such an agreement can be partly specified by *Document Type Definitions* and XML Schemas. Although DTDs and XML Schemas do not specify the data's meaning, they do specify the names of elements and attributes (the vocabulary) and their use in documents. Both are mechanisms with which you can specify the structure of XML documents. You can then validate specific documents against the structure prescription specified by a DTD or an XML Schema.

DTDs provide only a simple structure prescription: they specify the allowed nesting of elements, the elements' possible attributes, and the locations where normal text is allowed. For example, a DTD might prescribe that every

1094-7167/01/$10.00 © 2001 IEEE

person element must have a name attribute and may have a child element called phone whose content must be text. A DTD's syntax looks a bit awkward, but it is actually quite simple.

XML Schemas are a proposed successor to DTDs. The XML Schema definition is still a candidate recommendation from the W3C (World Wide Web Consortium), which means that, although it is considered stable, it might still undergo small revisions. XML Schemas have several advantages over DTDs. First, the XML Schema mechanism provides a richer grammar for prescribing the structure of elements. For example, you can specify the exact number of allowed occurrences of child elements, you can specify default values, and you can put elements in a *choice* group, which means that exactly one of the elements in that group is allowed at a specific location. Second, it provides data typing. In the example in the previous paragraph, you could prescribe the phone element's content as five digits, possibly preceded by another five digits between brackets. A third advantage is that the XML Schema definition provides inclusion and derivation mechanisms. This lets you reuse common element definitions and adapt existing definitions to new practices.

A final difference from DTDs is that XML Schema prescriptions use XML as their encoding syntax. **(**XML is a metalanguage, remember?) This simplifies tool development, because both the structure prescription and the prescribed documents use the same syntax. The XML Schema specification's developers exploited this feature by using an XML Schema document to define the class of XML Schema documents. After all, because an XML Schema prescription is an XML application, it must obey rules for its structure, which can be defined by another XML Schema prescription. However, this recursive definition can be a bit confusing.

RDF represents data about data

XML provides a syntax to encode data; the resource description framework is a mechanism to tell something about data. As its name indicates, it is not a language but a model for representing data about "things on the Web." This type of data about data is called *metadata*. The "things" are *resources* in RDF vocabulary.

RDF's basic data model is simple:

Table 1. An RDF description consisting of three triples indicating that a specific Web page was created by something with a name John and a phone number "47782."

Object	Attribute	Value
http://www.w3.org/	created_by	#anonymous_resource1
#anonymous_resource1	name	"John"
#anonymous_resource1	phone	"47782"

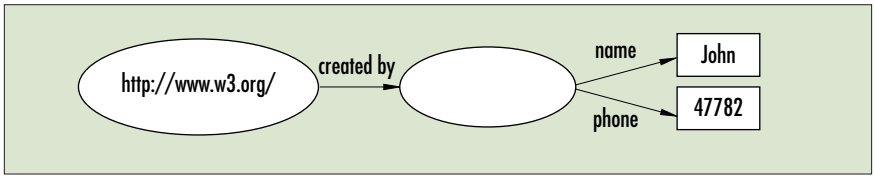


Figure 1. A directed labeled graph for the triples in Table 1.

besides resources, it contains *properties* and *statements*. A property is a specific aspect, characteristic, attribute, or relation that describes a resource. A statement consists of a specific resource with a named property plus that property's value for that resource. This value can be another resource or a *literal* value: free text, basically. Altogether, an RDF description is a list of triples: an object (a resource), an attribute (a property), and a value (a resource or free text). For example, Table 1 shows the three triples necessary to state that a specific Web page was created by something with a name "John" and a phone number "47782."

You can easily depict an RDF model as a directed labeled graph. To do this, you draw an oval for every resource and an arrow for every property, and you represent literal values as boxes with values. Figure 1 shows such a graph for the triples in Table 1.

These example notations reveal that RDF is ignorant about syntax; it only provides a model for representing metadata. The triple list is one possible representation, as is the labeled graph, and other syntactic representations are possible. Of course, XML would be an obvious candidate for an alternative representation. The specification of the data model includes such an XML-based encoding for RDF.

As with XML, an RDF model does not define (a priori) the semantics of any application domain or make assumptions about a particular application domain. It just provides a domain-neutral mechanism to describe metadata. Defining domain-specific properties and their semantics requires additional facilities.

Defining an RDF vocabulary: RDF Schema

Basically, RDF Schema is a simple type system for RDF. It provides a mechanism to define domain-specific properties and classes of resources to which you can apply those properties.

The basic modeling primitives in RDF Schema are *class* definitions and *subclass-of* statements (which together allow the definition of class hierarchies), *property* definitions and *subproperty-of* statements (to build property hierarchies), *domain* and *range* statements (to restrict the possible combinations of properties and classes), and *type* statements (to declare a resource as an instance of a specific class). With these primitives you can build a schema for a specific domain. In the example I've been using throughout this tutorial, you could define a schema that declares two classes of resources, Person and WebPage, and two properties, name and phone, both with the domain Person and range Literal. You could use this schema to define the resource http://www.w3.org/ as an instance of WebPage and the anonymous resource as an instance of Person. Together, this would give some interpretation and validation possibilities to the RDF data.

RDF Schema is quite simple compared to full-fledged knowledge representation languages. Also, it still does not provide exact semantics. However, this omission is partly intentional; the W3C foresees and advocates further extensions to RDF Schema.

Because the RDF Schema specification is also a kind of metadata, you can use RDF to encode it. This is exactly what occurs in the RDF Schema specification

document. Moreover, the specification provides an RDF Schema document that defines the properties and classes that the RDF Schema specification introduced. As with the XML Schema specification, such a recursive definition of RDF Schema looks somewhat confusing.

XML and RDF are different formalisms with their own purposes, and their roles in the realization of the Semantic Web vision will be different. XML aims to provide an easy-to-use syntax for Web data. With it, you can encode all kinds of data that is exchanged between computers, using XML Schemas to prescribe the data structure. This makes XML a fundamental language for the Semantic Web, in the sense that many techniques will probably use XML as their underlying syntax.

XML does not provide any interpretation of the data beforehand, so it does not contribute much to the "semantic" aspect of the Semantic Web. RDF provides a standard model to describe facts about Web resources, which gives some interpretation to the data. RDF Schema extends those interpretation possibilities somewhat more. However, to realize the Semantic Web vision, it will be necessary to express even more semantics of data, so further extensions are needed. There are already some initial steps in this direction—for example, the DAML+OIL (DARPA Agent Markup Language + Ontology Inference Layer) language, which adds new modeling primitives and formal semantics to RDF Schema.

The "Further Reading" sidebar contains pointers to more detailed explanations of XML and RDF and lists the URLs of the official homepages of XML, RDF, and the Semantic Web Activity at the W3C. Through those pages, you can find many projects and applications related to these topics. ■

Further Reading

- The pages at www.w3.org/XML and www.w3.org/RDF contain pointers to the official definitions of the languages that I covered in this minitutorial.
- XML.com (www.xml.com) contains technical introductions to both XML and XML Schemas.
- Pierre-Antoine Champin provides comprehensive tutorial on RDF and RDF Schema at www710.univ-lyon1.fr/~champin/rdf-tutorial.
- Robin Cover maintains a comprehensive online reference for XML and related techniques at www.oasis-open.org/cover.
- The vision of the Semantic Web is sketched at www.w3.org/2001/sw/Activity.
- The DAML+OIL extension to RDF Schema lives at www.daml.org/2001/03/daml+oil-index.

Michel Klein is a PhD student at the Information Management Group of the Vrije Universiteit in Amsterdam. His research interests include ontology modeling, maintenance, and integration, and representation and interoperability issues of semistructured data. Contact him at the Faculty of Sciences, Division of Mathematics and Computer Science, Vrije Universiteit, De Boelelaan 1081a, 1081 HV Amsterdam, Netherlands; michel.klein@cs.vu.nl; www.cs.vu.nl/~mcaklein.

IEEE COMPUTER SOCIETY

PURPOSE The IEEE Computer Society is the world's largest association of computing professionals, and is the leading provider of technical information in the field.

MEMBERSHIP Members receive the monthly magazine **COMPUTER**, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.

COMPUTER SOCIETY WEB SITE
The IEEE Computer Society's Web site, at **http://computer.org**, offers information and samples from the society's publications and conferences, as well as a broad range of information about technical committees, standards, student activities, and more.

OMBUDSMAN Members experiencing problems—magazine delivery, membership status, or unresolved complaints—may write to the ombudsman at the Publications Office or send an e-mail to membership@computer.org.

CHAPTERS Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.

AVAILABLE INFORMATION
To obtain more information on any of the following, contact the Publications Office:

- Membership applications
- Publications catalog
- Draft standards and order forms
- Technical committee list
- Technical committee application
- Chapter start-up procedures
- Student scholarship information
- Volunteer leaders/staff directory
- IEEE senior member grade application (requires 10 years practice and significant performance in five of those 10)

To check membership status or report a change of address, call the IEEE toll-free number, +1 800 678 4333. Direct all other Computer Society-related questions to the Publications Office.

PUBLICATIONS AND ACTIVITIES

Computer. An authoritative, easy-to-read magazine containing tutorial and in-depth articles on topics across the computer field, plus news, conferences, calendar, industry trends, and product reviews.

Periodicals. The society publishes 11 magazines and nine research transactions. Refer to membership application or request information as noted at left.

Conference Proceedings, Tutorial Texts, Standards Documents. The Computer Society Press publishes more than 150 titles every year.

Standards Working Groups. More than 200 of these groups produce IEEE standards used throughout the industrial world.

Technical Committees. Thirty TCs publish newsletters, provide interaction with peers in specialty areas, and directly influence standards, conferences, and education.

Conferences/Education. The society holds about 100 conferences each year and sponsors many educational activities, including computing science accreditation.

EXECUTIVE COMMITTEE

President:
BENJAMIN W. WAH *

University of Illinois
Coordinated Sci Lab
1308 W Main St
Urbana, IL 61801-2307
Phone: +1 217 333 3516
Fax: +1 217 244 7175
b.wab@computer.org

President-Elect:
WILLIS K. KING *

Past President:
GUYLAINE M. POLLOCK*

VP, Educational Activities:
CARL K. CHANG (1ST VP)*

VP, Conferences and Tutorials:
GERALD L. ENGEL*

VP, Chapters Activities:
JAMES H. CROSS †

VP, Publications:
RANGACHAR KASTURI †

VP, Standards Activities:
LOWELL G. JOHNSON*

VP, Technical Activities:
DEBORAH K. SCHERRER (2ND VP)*

Secretary:
WOLFGANG K. GILOI*

Treasurer:
STEPHEN L. DIAMOND*

2000–2001 IEEE Division V Director: DORIS L. CARVER †

2001–2002 IEEE Division VIII Director: THOMAS W. WILLIAMS †

Acting Executive Director:
ANNE MARIE KELLY†

* voting member of the Board of Governors
† nonvoting member of the Board of Governors

BOARD OF GOVERNORS

Term Expiring 2001: *Kenneth R. Anderson, Wolfgang K. Giloi, Haruhisa Ichikawa, Lowell G. Johnson, Ming T. Liu, David G. McKendry, Anneliese Amschler Andrews*

Term Expiring 2002: *Mark Grant, James D. Isaak, Gene F. Hoffnagle, Karl Reed, Kathleen M. Swigger, Ronald Waxman, Akihiko Yamada*

Term Expiring 2003: *Fiorenza C. Albert-Howard, Manfred Broy, Alan Clements, Richard A. Kemmerer, Susan A. Mengel, James W. Moore, Christina M. Schober*

Next Board Meeting: *25 May 2001, Seattle, Washington*

EXECUTIVE STAFF

Acting Executive Director : ANNE MARIE KELLY
Publisher: ANGELA BURGESS
Acting Director, Volunteer Services:
MARY-KATE RADA
Chief Financial Officer: VIOLET S. DOAN
Director, Information Technology & Services:
ROBERT CARE
Manager, Research & Planning: JOHN C. KEATON

COMPUTER SOCIETY OFFICES

Headquarters Office
1730 Massachusetts Ave. NW
Washington, DC 20036-1992
Phone: +1 202 371 0101 • Fax: +1 202 728 9614
E-mail: hq.ofc@computer.org

Publications Office
10662 Los Vaqueros Cir., PO Box 3014
Los Alamitos, CA 90720-1314
General Information:
Phone:+1 714 821 8380
E-mail: membership@computer.org
Membership and Publication Orders:
Phone: +1 800 272 6657
Fax: +1 714 821 4641
E-mail: cs.books@computer.org

European Office
13, Ave. de L'Aquilon
B-1200 Brussels, Belgium
Phone: +32 2 770 21 98 • Fax: +32 2 770 85 05
E-mail: euro.ofc@computer.org

Asia/Pacific Office
Watanabe Building
1-4-2 Minami-Aoyama, Minato-ku,
Tokyo 107-0062, Japan
Phone: +81 3 3408 3118 • Fax: +81 3 3408 3553
E-mail: tokyo.ofc@computer.org

IEEE OFFICERS

President:
JOEL B. SNYDER

President-Elect:
RAYMOND D. FINDLAY

Executive Director:
DANIEL J. SENESE

Secretary:
DAVID J. KEMP

Treasurer:
DAVID A. CONNOR

VP, Educational Activities:
LYLE D. FEISEL

VP, Publications Activities:
MICHAEL S. ADLER

VP, Regional Activities:
ANTONIO BASTOS

VP, Standards Association:
DONALD C. LOUGHRY

VP, Technical Activities:
ROBERT A. DENT

President, IEEE-USA:
NED R. SAUTHOFF

Agents and the Semantic Web

James Hendler, *University of Maryland*

At a colloquium I attended recently, a speaker described a "science fiction" vision comprising agents running around the Web performing complex actions for their users. The speaker argued that we are far from the day this vision would become a reality because we don't have the infrastructure to make it happen.

Many challenges of bringing communicating multiagent systems to the Web require ontologies. The integration of agent technology and ontologies could significantly affect the use of Web services and the ability to extend programs to perform tasks for users more efficiently and with less human intervention.

Although I agree with his assessment about infrastructure, his claim that we are "far from the day" is too pessimistic. A crucial component of this infrastructure, a standardized Web ontology language, is emerging. This article offers a few pointers to this emerging area and shows how the ontology languages of the Semantic Web can lead directly to more powerful agent-based approaches—that is, to the realization of my colleague's "science fiction" vision.

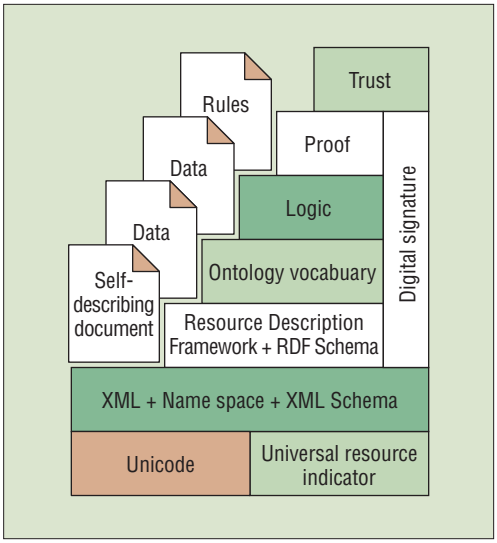


Figure 1. The Semantic Web "layer cake" presented by Tim Berners-Lee at the XML 2000 conference.

What is an ontology, really?

There are a number of terms we sometimes abuse in the AI community. These terms become even more confusing when we interact with other communities, such as Web toolkit developers, who also abuse them. One such term is *ontology*, which the *Oxford English Dictionary* defines as "the science or study of being." In AI, we usually attribute the notion of ontology to, essentially, the specification of a conceptualization—that is, defined terms and relationships between them, usually in some formal and preferably machine-readable manner.[1] Even more complicated is the relationship between ontologies and logics. Some people treat ontology as a subset of logic, some treat logic as a subset of ontological reasoning, and others consider the terms disjoint.

In this article, I employ the term as it is currently being used in Semantic Web circles. I define ontology as a set of knowledge terms, including the vocabulary, the semantic interconnections, and some simple rules of inference and logic for some particular topic. For example, the ontology of cooking and cookbooks includes ingredients, how to stir and combine the ingredients, the difference between simmering and deep-frying, the expectation that the products will be eaten or drunk, that oil is for cooking or consuming and not for lubrication, and so forth.

In practice, it is useful to consider more complex logics and inference systems to be separate from an

1094-7167/01/$10.00 © 2001 IEEE

ontology. Figure 1, derived from a talk given by Tim Berners-Lee at the recent XML 2000 conference, shows the proposed layers of the Semantic Web with higher-level languages using the syntax and semantics of lower levels. This article focuses primarily on the ontology language level and the sort of agent-based computing that ontology languages enable. Higher levels (with complex logics and the exchange of proofs to establish trust relationships) will enable even more interesting functionality, but I've left those to be discussed in other articles.

Semantic Web ontologies

The Semantic Web, as I envision it evolving, will not primarily consist of neat ontologies that expert AI researchers have carefully constructed. I envision a complex Web of semantics ruled by the same sort of anarchy that rules the rest of the Web. Instead of a few large, complex, consistent ontologies that great numbers of users share, I see a great number of small ontological components consisting largely of pointers to each other. Web users will develop these components in much the same way that Web content is created.

In the next few years, almost every company, university, government agency, or ad hoc interest group will want their Web resources linked to ontological content because of the many powerful tools that will be available for using that content. Information will be exchanged between applications, letting programs collect and process Web content and exchange information freely. On top of this infrastructure, agent-based computing will become much more practical. Distributed computer programs interacting with nonlocal Web-based resources might eventually become the dominant way in which computers interact with humans and each other. Such interaction will also be a primary means of computation in the not-so-distant future.

However, for this vision to become a reality, a phenomenon similar to the Web's early days must occur. Web users will not mark up their Web pages unless they perceive value in doing so, and tools to demonstrate this value will not be developed unless Web resources are marked up. To help solve this chicken-and-egg problem, DARPA is funding a set of researchers to both develop freely available tools and provide significant content for these tools to manipulate. This should demonstrate to the government and other parts of society that the Semantic Web can be a reality.

But without some killer apps showing the great power of Web semantics, it will still be a long row to hoe. Although I don't claim to have all the answers, perhaps some ideas in the remainder of this article will inspire the creation of exciting Web-agent applications. I will develop this vision one step at a time by describing the creation of pages with ontological information, the definition of services in a machine-readable form, and the use of logics and agents that provide important new capabilities.

Markup for free

A crucial aspect of creating the Semantic Web is to enable users who are not logic experts to create machine-readable Web content. Ideally, most users shouldn't even need to know that Web semantics exist. Lowering markup's cost isn't enough; for many users it must be free. Semantic markup should be a by-product of normal computer use. Much like current Web content, a small number of tool creators and Web ontology designers will need to know the details, but most users will not even know ontologies exist.

The ability to link and browse ontological relations enabled by the Web's use of semantics will be a powerful tool for users who do know what ontologies are and why they should be used.

Consider any of the well-known products for creating online slide shows. Several of these products contain libraries of clippings that you can insert into a presentation. Software developers could mark these clippings with pointers to ontologies. The save-as-HTML feature could include linking these products to their respective ontologies. So, a presentation that had pictures of, for example, a cow and a donkey would be linked to barnyard animals, mammals, animals, and so forth. While doing so would not guarantee appropriate semantics—the cow might be the mascot of some school or the donkey the icon of some political party—retrieval engines could use the markups as clues to what the presentations contain and how they can be linked to other ones. The user simply creates a slide show, but the search tools do a better job of finding results.

An alternative example is a markup tool driven by one or more ontologies. Consider a page-creation tool that represents hierarchical class relations as menus. Properties of the classes could be tied to various types of forms, and these made available through simple Web forms. A user could thus choose from a menu to add information about a person, and then choose a relative (as opposed to a friend, professional acquaintance, and so forth) and then a daughter. The system would use the semantics to retrieve the properties of daughters specified in the ontologies and to display them to the user as a form to be filled out with strings (such as `name`) or numbers (`age`)—or to browse for related links (`homepage`), online images (`photo-of`), and so forth. The system would then lay these out using appropriate Web page design tools while recording the relevant instance information.

Because the tool could be driven by any ontology, libraries of terms could be created (and mixed) in many different ways. Thus, a single easy-to-use tool would allow the creation of homepages (using ontologies on people, hobbies, and so forth), professional pages (using ontologies relating to specific occupations or industries), or agency-specific pages (using ontologies relating to specific functions). In an easy, interactive way the tool would help a user create a page and would provide free markup. Also, mixtures of the various ontologies and forms could be easily created, thus helping to create the Semantic Web of pages linking to many different ontologies, as I mentioned earlier.

Incremental ontology creation

Not only can pages be created with links to numerous ontologies, but the ontologies can also include links between them to reuse or change terms. The notion of creating large ontologies by combining components is not unique to the Semantic Web vision.[2] However, the ability to link and browse ontological relations enabled by the Web's use of semantics will be a powerful tool for users who do know what ontologies are and why they should be used.

How will it all work? Consider Mary, the Webmaster for a new business-to-consumer Web site for an online pet shop. Browsing

Query Processed:

- A satellite image taken yesterday at 10 AM is available on the Web at http://….
- A new satellite image, to be taken today at 10 AM, will be available for $100—click here to authorize transfer of funds and obtain image. (You will need a valid credit card number from one of the following providers.…)
- In an emergency situation, a Coast Guard observer plane can be sent to any location within the area you indicate. Service Note: You will be responsible for cost of flight if the situation does not result in an emergency pickup. Click here for more information.
- A high-altitude observer can be sent to your location in 13 hours. Click here to initiate procedure. (You will need to provide US military authorization, a valid military unit code, and the name of the commanding officer. Abuse of this procedure can result in fine or imprisonment.)
- A service entitled commercial service for providing satellite images is advertised as becoming available in 2004. See http://… for more information.

Figure 2. The results of processing a fictitious agent-based query from a fishing vessel that finds itself in a difficult weather situation.

through a Web ontology repository (such as the one at www.daml.org/ontologies/), she finds that many interesting ontologies are available. Selecting a product ontology, Mary uses a browser to choose the various classes and relations that she wants to include in her ontology. Several of these might need to be further constrained depending on the properties of her particular business. For example, Mary must define some of these properties for the various animals she will sell.

Searching further in the repository, Mary finds a biological taxonomy that contains many classes, such as **feline**, **canine**, **mammal**, and **animal**. She finds that these ontologies contain several properties relevant to her business, so she provides links to them. She adds a new descriptor field to **animal** called **product shipping type** and sets it to default to the value **alive** (not a standard property or default in the product ontology she chose to extend).

Finally, she notices that although the biological ontology contains several kinds of felines, it didn't use the categories she wanted (popular pets, exotic pets, and so forth), so she adds these classes as subclasses of the ones in the parent ontology and defines their properties. Saving this ontology on her Web site, she can now use other ontology-based tools to organize and manage her Web site. Mary is motivated to add the semantics to her site by both these tools and the other powerful browsing and search tools that the semantics enable.

The many ontology-based search and browsing tools on the Web, when pointed at her pages, can use this information to distinguish her site from the non-ontology-based sites that her competitors run. This makes it easy for her to extend her site to use various business-to-business e-commerce tools that can exploit Web ontologies for automated business uses. In addition, she might submit her ontology back into one of the repositories so that others in her profession can find it and use it for their own sites. After all, the power of the ontologies is in the sharing; the more people using common terms with her, the better.

Ontologies and services

Web services might be one of the most powerful uses of Web ontologies and will be a key enabler for Web agents. Recently, numerous small businesses, particularly those in supply chain management for business-to-business e-commerce, have been discussing the role of ontologies in managing machine-to-machine interactions. In most cases, however, these approaches assume that computer program constructors primarily use ontologies to ensure that everyone agrees on terms, types, constraints, and so forth. So, the agreement is recorded primarily offline and used in Web management applications. On the Semantic Web, we will go much further than this, creating machine-readable ontologies used by "capable" agents to find these Web services and automate their use.

A well-known problem with the Web is that finding the many available Web services is difficult. For example, when I first started writing this article, I wanted to send a Web greeting card but didn't know the name of any companies offering such a service. Using standard keyword-based searches did not help much. The query "web greeting card" turned up many links to sites displaying greeting cards or using the terms on their pages. In fact, for these three keywords, several of the most common search engines did not turn up the most popular Web greeting card service provider in their top 20 suggestions. A search on "eCards" would have found the most popular site, but I didn't happen to know this particular neologism.

As I'm finalizing this article, the search engines are now actually finding the most popular site with the "web greeting card" keywords. However, if I want something more complex—for example, an anniversary card for my mother-in-law that plays "Hava Nagila"—I'm still pretty much out of luck. As the number of services grows and the specificity of our needs increases, the ability of current search engines to find the most appropriate services is strained to the limit.

Several efforts are underway to improve this situation. Some examples are the Universal Description, Discovery, and Integration specification (www.uddi.org); ebXML (www.ebXML.org); and eSpeak (www.e-speak.hp.com). These efforts focus on *service advertisements*. By creating a controlled vocabulary for service advertisements, search engines could find these Web services. So, Mary's pet site (discussed above) might have an annotation that it provides a "sell" service of object "pet," which would let pet buyers find it more easily. Similarly, a Web greeting card site could register as something such as "personal service, e-mail, communications," and a user could more easily get to it without knowing the term "eCard."

Semantic Web techniques can—and must—go much further. The first use of ontologies on the Web for this purpose is straightforward. By creating the service advertisements in an ontological language, you would be able to use the hierarchy (and property restrictions) to find matches through class and subclass properties or other semantic links. For example, someone looking to buy roses might find florists (who sell flowers) even if no exact match served the purpose. Using description logic (or other inferential means), the user could even find categorizations that weren't explicit. So, for example, specifying a search for animals that were of "size = small" and "type = friendly," the user could end up finding the pet shop Mary is working for, which happens to be overflowing in hamsters and gerbils.

However, by using a combination of Web pointers, Web markup, and ontology languages, we can do even better than just

putting service advertisements into ontologies. By using these techniques we can also include a machine-readable description of a service (as to how it runs) and some explicit logic describing the consequences of using the service. Such service descriptions and service logic will lead us to the integration of agents and ontologies in some exciting ways.

Agents and services

In an earlier article, I described a vision of intelligent Web agents using the analogy of travel agents.[3] Rather than doing everything for a user, the agents would find possible ways to meet user needs and offer the user choices for their achievement. Much as a travel agent might give you a list of several flights to take, or a choice of flying as opposed to taking a train, a Web agent could offer several possible ways to get you what you need on the Web.

Consider a Web-enabled method for saving the doomed crew of *The Perfect Storm*.[4] In this story, now a major motion picture, a crew of fishermen is out at sea when weather conditions conspire to create a storm of epic proportions. For various reasons, the crew is unable to get a detailed weather map, so they miss that the storm is developing right in their way. Instead of avoiding it, they end up at its center, with tragic results.

How could Web agents have helped? As the ship's captain goes to call land, a wave hits and his cell phone is swept overboard. Luckily, he is a savvy Web user and has brought his wireless Web device with him. Checking the weather forecast from a standard weather site, he determines that a storm is coming, but he does not find enough detail for his needs. He goes to an agent-enabled geographical server site and invokes the query "Get me a satellite photo of this region of the Atlantic," and he draws a box on an appropriate map.

The system comes back a little later with the message shown in Figure 2. Options range from a picture available on the Web (possibly out of date) to other services (that might need special resources) and even future options being announced. The captain now chooses an option on the basis of what available resources he has and what criterion he is willing to accept. Recognizing the gravity of his situation, he invokes the Coast Guard option, which creates a scheduled overflight for his GPS location. Seeing the emerging weather, the Coast Guard arranges an emergency pickup at sea, and the sailors can go on to fish again some other day.

Using the tools of the Semantic Web, we can make this sort of thing routine and available to anyone who needs to use a Web service for any purpose. We simply need to make expressive service capability advertisements available to, and usable by, Web agents. Figure 3 depicts a complete instance of a potential *service class*. Each service class has three properties: a pointer to the service advertisement as discussed above, a pointer to a service description, and a declarative service logic. I will discuss the service logic later; I first want to concentrate on service descriptions.

Consider visiting a current business-to-consumer Web site, such as a book vendor. When you are ready to order, you usually have to fill out a form. When you click on the Submit button, you're taken to another form or returned to the same form to provide missing information or to fix an error. When you pass through the first form, you get directed to a new form where the same might happen, until eventually you provide the information necessary to complete the order. Most other Web services require similar interactions, whether to buy an item, get directions to a location, or find a particular image.

The most common way to develop these systems is with the Common Gateway Interface (CGI), in which procedural code is written to invoke various functions of the Web protocols. This code links the set of Web pages to an external resource, which means that the invocation procedure is represented procedurally on the Web. Thus, an agent visiting the page cannot easily determine the set of information that must be provided or analyze other features of the code.

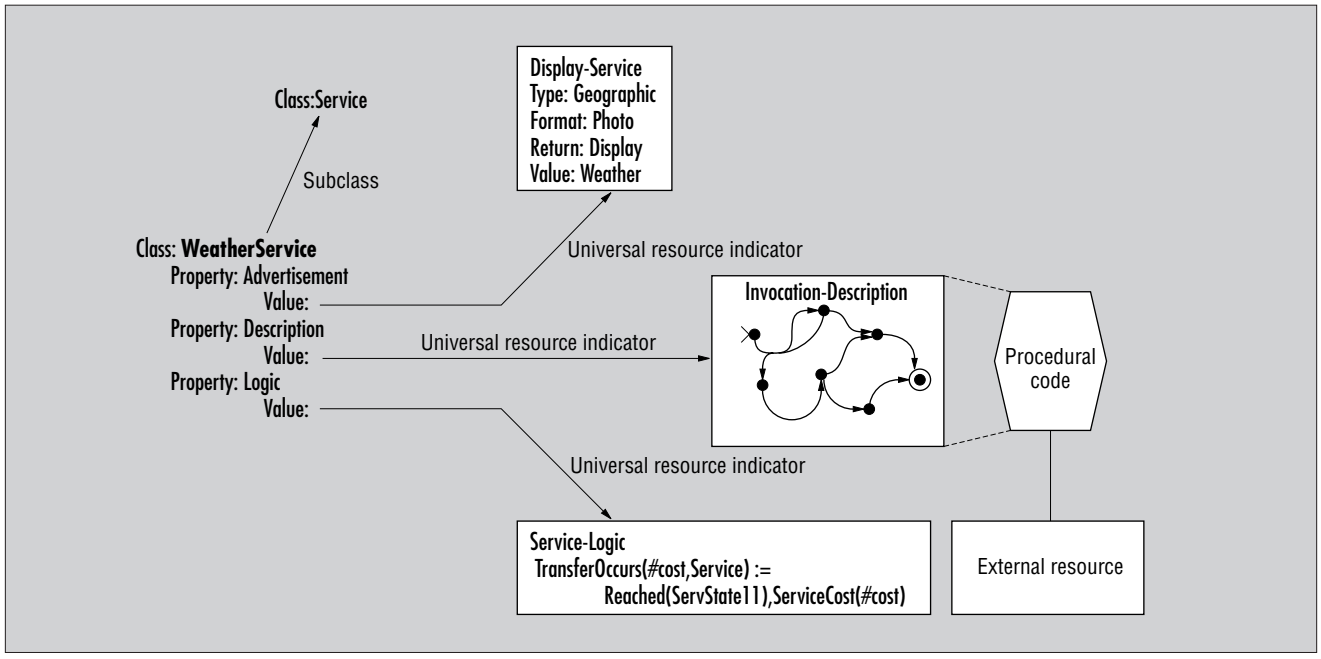


Figure 3. A potential service class and its properties on the Semantic Web.

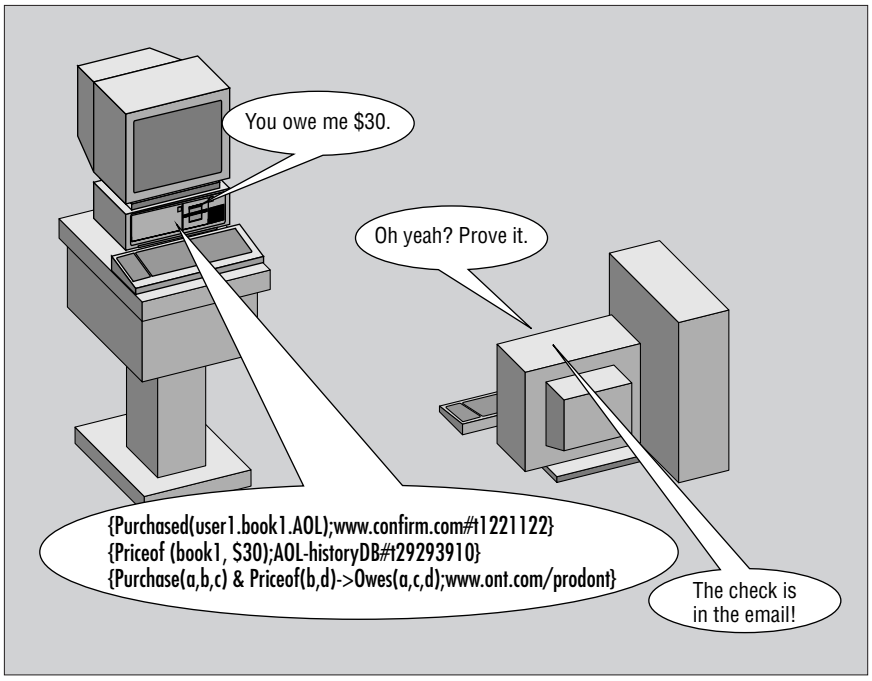


Figure 4. Agents exchanging simple proofs.

On the Semantic Web, solving this problem will be easy by using a declarative framework. Eventually you might wish to use some sort of Web-enabled logic language, but there is a much simpler way to get started. Figure 3 shows the invocation of the procedural code through a simple finite-state automaton. An ontology language such as DAML+OIL (see the sidebar "DAML and Other Languages") could be easily used to define an ontology—not of services but of the terms needed to describe the invocation of services.

Using the example of a finite-state machine (FSM), we can see what this ontology would contain. It would start with classes such as State and Link and have special subclasses such as StartState and EndState. Constraints and properties would be described to give links a head and tail, to give states a list of the links that lead out from them, and to give states a name, URI (universal resource identifier), or other identifying property. This would provide a base ontology that specific types of service providers could extend (much as Mary extended a biological ontology in the earlier example), and in which specialized ontologies could easily describe sets of terms for general use.

For example, a "standard Web sale" could be defined in some service ontology comprising a particular set of states and links. A service provider could then simply say that a particular part of a transaction was a standard Web sale, which would then find the necessary set of links and nodes through a pointer on the Web.

Exciting capabilities arise through creating such ontologies. Because these ontologies are Web-enabled and declarative, agents coming to a page containing a service description could analyze the FSM found there and would be able to determine the particular information needs for invoking the service (and reaching an EndState). An agent that had access to a set of information about a user could analyze the FSM and determine if that information would be sufficient for using this service. If not, the agent could inform the user as to what additional information would be required or other action taken.

While I've described primarily an FSM approach, there is no reason this couldn't be done using any other declarative framework. More expressive logic languages or other declarative frameworks would extend the capabilities of agents to analyze the information needs, resource requirements, and processing burden of the services so described. As these languages are linked to CGI scripts or other procedural techniques, the agents could perform the procedural invocation. This would let them actually run the services (without user intervention), thus allowing a very general form of agent interaction with off-Web resources.

Service logics

By defining languages that let users define structural ontologies, current projects (including the DARPA DAML initiative) are exploring the extension of Web ontologies to allow rules to be expressed within the languages themselves. These efforts vary in the complexity of the rules allowed, and range from description logics (as in the DAML+OIL language mentioned earlier), to SHOE's use of Horn-clause-like rules,[5] and even to first- and higher-order logics in several exploratory efforts.[6–9]

Whatever types of rules you use, they can be particularly effective in connection with the service classes, as Figure 3 shows. The service class contains (in addition to the service advertisement and service description) a pointer to a URI containing associated service logic. This logic can be used to express information that goes beyond the information contained in the service description.

For example, returning to the agent replies in Figure 2, consider a case in which the service offers an up-to-date picture (to be taken tomorrow) at some particular cost. A rule such as

```
TransferOccurs(#cost,Service) :=
    Reached(ServState11), ServiceCost(#cost)
```

might represent the information that the actual transfer of funds will occur upon reaching a particular point in the service invocation (ServState11 in this case). This information would not be obvious from the state machine itself but could be useful in several kinds of e-commerce transactions. For example, users often leave a site without completing a particular CGI script, and they cannot always know whether they've actually completed a transaction and incurred a credit card charge. Using service logics, such things could be made explicit.

More interesting transactional logics might also be used. Figure 4 shows a potential interaction between two Web agents that can use proof checking to confirm transactions. An agent sends an annotated proof to another agent. The annotations can be pointers to a particular fact on the Web or to an ontology where a particular rule resides. The agent receiving this proof can analyze it, check the pointers (or decide they are trusted

by some previous agreements), and check that the ontology is one it can read and agree with. This lets the agent recognize that a valid transaction has occurred and allow the funds to be transferred.

Such service logics could serve many other purposes as well. For example, *Heterogeneous Agent Systems*[10] discusses the use of deontic logics and agent programs for multiagent systems. These logics, tied to the appropriate service descriptions, can represent what an agent can do and when it can or cannot do so. Logical descriptions of services could also be used for automated matchmaking and brokering, for planning a set of services that together achieve a user's goal, and for other capabilities currently discussed (but not yet implemented) for multiagent systems.

Agent-to-agent communication

Of course, having pages, service descriptions, and agent programs that are linked to many ontologies, which might themselves include links to still other ontologies and so on, introduces some compelling issues. Figure 5 shows a representation of a small piece of this ontological Web. The small boxes represent agents or other Web resources that use the terms in Web ontologies represented by the larger boxes. The arrows represent any mechanism that provides a mapping (full or partial) from one ontology to another. This mapping can be as simple as inclusion of terms or as complex as some sort of ad hoc mapping program that simply reads in terms from one and spits out terms of another. The figure shows one DAG (directed acyclic graph) that could be taken from the much larger Web of ontologies.

Assuming agents are communicating with each other using the terms in these ontologies for the content terms, it is relatively straightforward for them to communicate. By linking to these ontologies, the agents commit to using the terms consistently with the usage mandated in that ontology. If the ontology specifies that a particular class has a particular property and that the property has some restriction, then each agent can assume that the other has legal values for that property maintaining that restriction.

What is more interesting, agents that are not using the same ontologies might still be able to communicate. If all mappings were perfect, then obviously any agent could communicate with any other by finding a common ontology they could both map into. More likely, however, is that the ontologies are only partially or imperfectly mapped. This would happen, for example, with Mary's pet shop site. When Mary defined her site's ontology as linking back to the zoo's animal ontology, she changed some definitions but left others untouched. Those terms that were not modified, or were modified in

DAML and Other Languages

The modern IT world is a dynamically changing environment with an exponentially increasing ability to create and publish data that rapidly swamps human abilities to process that data into information. Agent-based computing can potentially help us recognize complex patterns in this widely distributed, heterogeneous, uncertain information environment. Unfortunately, this potential is hampered by the difficulty agents face in understanding and interacting with data that is either unprocessed or in natural languages. The inability of agents to understand the conceptual aspects of a Web page, their difficulty in handling the semantics inherent in program output, and the complexity of fusing sensor output information—to name but a few problems—truly keep the agent revolution from happening.

One potential solution is for humans to meet the computer halfway. By using tools to provide markup annotations attached to data sources, we can make information available to agents in new and exciting ways. The goal of the DARPA Agent Markup Language (DAML) program is to develop a language aimed at representing semantic relations in machine-readable ways that will be compatible with current and future Internet technologies. The program is currently developing prototype tools to show the potential of such markups to provide revolutionary capabilities that will change the way humans interact with information.

To realize these goals, Internet markup languages must move beyond the implicit semantic agreements inherent in XML and community-specific controlled languages. DARPA is leading the way with DAML, which will be a semantic language that ties the information on a page to machine-readable semantics. The language must allow for communities to extend simple ontologies for their own use, allowing the bottom-up design of meaning while allowing sharing of higher-level concepts. In addition, the language will provide mechanisms for the explicit representation of services, processes, and business models so as to allow nonexplicit information (such as that encapsulated in programs or sensors) to be recognized.

DAML will provide a number of advantages over current markup approaches. It will allow semantic interoperability at the level we currently have syntactic interoperability in XML. Objects in the Web can be marked (manually or automatically) to include descriptions of information they encode, descriptions of functions they provide, and descriptions of data they can produce. Doing so will allow Web pages, databases, programs, models, and sensors all to be linked together by agents that use DAML to recognize the concepts they are looking for. If successful, information fusion from diverse sources will become a reality.

DARPA funds work in the development of DAML to help the US military in areas of command and control and for use in military intelligence. For example, one use of DAML is to improve the organization and retrieval of large military information stores such as those at the US Center for Army Lessons Learned. With respect to intelligence, DAML is aimed at improving the integration of information from many sources to provide specific indications and warnings aimed at preventing terrorist attacks on military targets such as last year's attack on the USS Cole in Yemen.

Recently, an ad hoc group of researchers formed the Joint US–EU committee on Agent Markup Languages and released a new version of DAML called DAML+OIL. This language is based on the Resource Description Framework (www.w3.org/rdf); you can find discussion of RDF's features on an open mailing list archived at http://lists.w3.org/Archives/Public/www-rdf-logic. For details of the language, a repository of numerous ontologies and annotated Web pages, and a full description of DAML and related projects see www.daml.org.

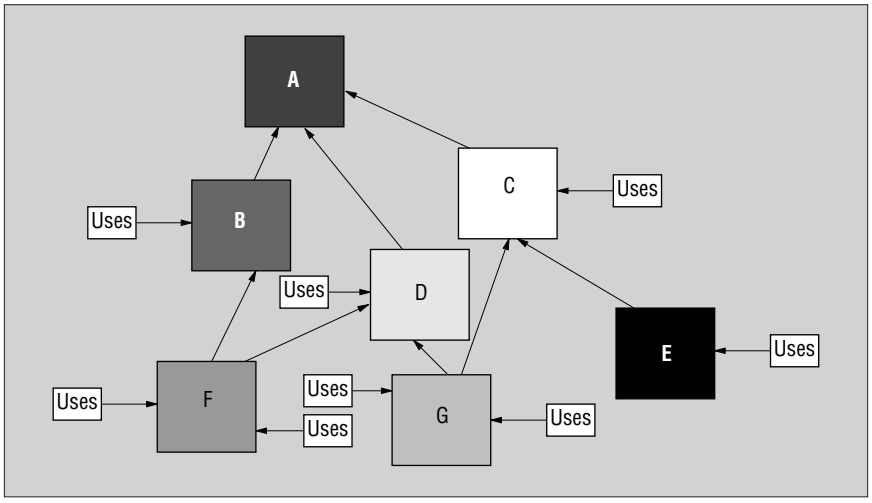


Figure 5. Mappings between agents and the ontologies they use.

certain restricted ways, could be mapped even if others couldn't. So, those ontologies made by combination and extension of others could, in principle, be partially mapped without too much trouble.

With this in mind, let's reconsider the DAG in Figure 5. Clearly, many of these agents could be able to find at least some terms that they could share with others. For agents such as those pointing at ontologies C and E, the terms they share might be some sort of subset. In this case the agent at E might be able to use only some of the terms in C (those that were not significantly changed when E was defined). Other agents, such as the ones pointing at F and G, might share partial terms from another ontology that they both changed (D in this case). In fact, all of the agents might share some terms with all the others, although this might take several mappings (and thus there might be very few common terms, if any, in some cases).

The previous discussion is purposely vague regarding what these mappings are and how they work. For certain kinds of restricted mappings, we might be able to obtain some interesting formal results. For example, if all mappings are inclusion links—that is, the lower ontology includes all the terms from the upper one in Figure 5—and we can find a rooted DAG among a set of agents, then we could guarantee that all those agents will share some terms with all others (although, in the worst case, some might only share the terms from the uppermost ontology). If the mappings are more ad hoc—they might, for example, be some sort of procedural maps defined by hand—we might lose provable properties but gain power or efficiency.

The research issues inherent in such ontology mappings are quite interesting and challenging. Two agents that communicate often might want to have maximal mappings or even a merged ontology. Two agents that are simply sending a single message (such as the invocation of an online service) might want some sort of quick on-the-fly translation limited to the terms in a particular message. Another approach might be to use very large ontologies, such as CYC,[11] to infer mapping terms between agents in other ontologies. The possibilities are endless and are another exciting challenge for researchers interested in bringing agents to the Semantic Web.

I did not intend this article to be a comprehensive technical tome. Rather, I hope that I have convinced you that several strands of research in AI, Web languages, and multiagent systems can be brought together in exciting and interesting ways.

Many of the challenges inherent in bringing communicating multiagent systems to the Web require ontologies of the type being developed in DARPA's DAML program and elsewhere. What is more important, the integration of agent technology and ontologies might significantly affect the use of Web services and the ability to extend programs to perform tasks for users more efficiently and with less human intervention.

Unifying these research areas and bringing to fruition a Web teeming with complex, intelligent agents is both possible and practical, although a number of research challenges still remain. The pieces are coming together, and thus the Semantic Web of agents is no longer a science fiction future. It is a practical application on which to focus current efforts. ❐

Acknowledgments

This paper benefited from reviews by a wide number of early readers. I especially thank Oliver Selfridge, who offered a comprehensive review, including the addition of several paragraphs. I also thank David Ackley, Tim Berners-Lee, Dan Brickley, Dan Connolly, Jeff Heflin, George Cybenko, Ora Lassila, Deborah McGuinness, Sheila McIlraith, Frank van Harmelen, Dieter Fensel, and

For Further Reading

Web sites

W3C Semantic Web Activity: www.w3.org/2001/sw
The DAML project: www.daml.org
The SHOE project: www.cs.umd.edu/projects/plus/SHOE
The Semantic Web Community Portal: www.semanticweb.org

Articles

J. Heflin, J. Hendler, and S. Luke, "Reading between the Lines: Using SHOE to Discover Implicit Knowledge from the Web," *Proc. AAAI-98 Workshop AI and Information Integration*, AAAI Press, Menlo Park, Calif., 1998, www.cs.umd.edu/projects/plus/SHOE/pubs/shoe-aaai98.ps.

S. McIlraith, "Modeling and Programming Devices and Web Agents," to be published in *Proc. NASA Goddard Workshop Formal Approaches to Agent-Based Systems*, Springer-Verlag, New York.

F. Zini and L. Sterling, "Designing Ontologies for Agents," *Proc. Appia-Gulp-Prode 99: Joint Conf. Declarative Programming*, 1999, pp. 29–42.

many participants in the CoABS, DAML, and TASK DARPA initiatives who offered comments on earlier drafts. Finally, I am indebted to an anonymous reviewer who, shall we say, wasn't impressed by an earlier version of this article and demanded copious changes. I made many of these changes, which improved the article greatly.

References

1. T.R. Gruber, "A Translation Approach to Portable Ontologies," *Knowledge Acquisition*, vol. 5, no. 2, 1993, pp. 199–220.
2. P. Clark and B. Porter, "Building Concept Representations from Reusable Components," *Proc. 14th Nat'l Conf. Artificial Intelligence* (AAAI-97), MIT Press, Cambridge, Mass., 1997, pp. 369–376.
3. J. Hendler, "Is There an Intelligent Agent in Your Future?" *Nature*, 11 Mar. 1999, www.nature.com/nature/webmatters/agents/agents.html (current 19 Mar. 2001).
4. S. Junger, *The Perfect Storm: A True Story of Men against the Sea*, W.W. Norton and Co., London, 1997.
5. J. Heflin and J. Hendler, "Dynamic Ontologies on the Web," *Proc. 17th Nat'l Conf. Artificial Intelligence* (AAAI 2000), MIT Press, Cambridge, Mass., 2000, pp. 443–449.
6. A.W. Appel and E.W. Felten, "Proof-Carrying Authentication," *Proc. 6th ACM Conf. Computer and Communications Security*, ACM Press, New York, 1999; www.cs.princeton.edu/~appel/papers/fpcc.pdf.
7. D. Fensel et al., "The Component Model of UPML in a Nutshell," *Proc. 1st Working IFIP Conf. Software Architecture* (WICSA 1), Kluwer Academic Publishers, 1999, ftp.aifb.uni-karlsruhe.de/pub/mike/dfe/paper/upml.ifip.pdf.
8. D. Fensel et al., "OIL in a Nutshell," *Proc. 12th European Workshop Knowledge Acquisition, Modeling, and Management* (EKAW-00), Springer-Verlag, New York, 2000; www.few.vu.nl/~frankh/postscript/EKAW00.pdf.
9. M. Genesereth et al., *Knowledge Interchange Format Version 3.0 Reference Manual*, http://logic.stanford.edu/kif/Hypertext/kif-manual.html.
10. V.S. Subrahmanian et al., *Heterogeneous Agent Systems*, MIT Press, Cambridge, Mass., 2000.
11. D. Lenat and R. Guha, *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project*, Addison-Wesley, Reading, Mass., 1990.

The Author

James Hendler is the Chief Scientist of DARPA's Information Systems Office and the program manager responsible for agent-based computing. He is on leave from the University of Maryland where he is a professor and head of both the Autonomous Mobile Robots Laboratory and the Advanced Information Technology Laboratory. He has joint appointments in the Department of Computer Science, the Institute for Advanced Computer Studies, the Institute for Systems Research, and is also an affiliate of the Electrical Engineering Department. He received a PhD in artificial intelligence from Brown University. Hendler received a 1995 Fulbright Foundation Fellowship, is a Fellow of the American Association for Artificial Intelligence, and is a member of the US Air Force Science Advisory Board. Contact him at jhendler@darpa.mil; www.cs.umd.edu/~hendler.

OIL: An Ontology Infrastructure for the Semantic Web

Dieter Fensel and Frank van Harmelen, *Vrije Universiteit, Amsterdam*
Ian Horrocks, *University of Manchester, UK*
Deborah L. McGuinness, *Stanford University*
Peter F. Patel-Schneider, *Bell Laboratories*

Ontologies play a major role in supporting information exchange across various networks. A prerequisite for such a role is the development of a joint standard for specifying and exchanging ontologies. The authors present OIL, a proposal for such a standard.

Researchers in artificial intelligence first developed *ontologies* to facilitate knowledge sharing and reuse. Since the beginning of the 1990s, ontologies have become a popular research topic, and several AI research communities—including knowledge engineering, natural language processing, and knowledge representation—have investigated them. More recently, the notion of an ontology is becoming widespread in fields such as intelligent information integration, cooperative information systems, information retrieval, electronic commerce, and knowledge management. Ontologies are becoming popular largely because of what they promise: a shared and common understanding that reaches across people and application systems.

Currently, ontologies applied to the World Wide Web are creating the *Semantic Web*.[1] Originally, the Web grew mainly around HTML, which provides a standard for structuring documents that browsers can translate in a canonical way to render those documents. On the one hand, HTML's simplicity helped spur the Web's fast growth; on the other, its simplicity seriously hampered more advanced Web applications in many domains and for many tasks. This led to XML (see Figure 1), which lets developers define arbitrary domain- and task-specific extensions (even HTML appears as an XML application—XHTML).

XML is basically a defined way to provide a serialized syntax for tree structures—it is an important first step toward building a Semantic Web, where application programs have direct access to data semantics. The *resource description framework*[2] has taken an important additional step by defining a syntactical convention and a simple data model for representing machine-processable data semantics. RDF is a standard for the Web metadata the World Wide Web Consortium (www.w3c.org/rdf) develops, and it defines a data model based on triples: object, property, and value. The *RDF Schema*[3] takes a step further into a richer representation formalism and introduces basic ontological modeling primitives into the Web. With RDFS, we can talk about classes, subclasses, subproperties, domain and range restrictions of properties, and so forth in a Web-based context. We took RDFS as a starting point and enriched it into a full-fledged Web-based ontology language called OIL.[4] We included these aspects:

- A more intuitive choice of some of the modeling primitives and richer ways to define concepts and attributes.
- The definition of a formal semantics for OIL.
- The development of customized editors and inference engines to work with OIL.

1094-7167/01/$10.00 © 2001 IEEE

Ontologies: A revolution for information access and integration

Many definitions of ontologies have surfaced in the last decade, but the one that in our opinion best characterizes an ontology's essence is this: "An ontology is a formal, explicit specification of a shared conceptualization."[5] In this context, *conceptualization* refers to an abstract model of some phenomenon in the world that identifies that phenomenon's relevant concepts. *Explicit* means that the type of concepts used and the constraints on their use are explicitly defined, and *formal* means that the ontology should be machine understandable. Different degrees of formality are possible. Large ontologies such as WordNet (www.cogsci.princeton.edu/~wn) provide a thesaurus for over 100,000 terms explained in natural language. On the other end of the spectrum is CYC (www.cyc.com), which provides formal axiomating theories for many aspects of commonsense knowledge. *Shared* reflects the notion that an ontology captures consensual knowledge—that is, it is not restricted to some individual but is accepted by a group.

The three main application areas of ontology technology are knowledge management, Web commerce, and electronic business.

Knowledge management

KM is concerned with acquiring, maintaining, and accessing an organization's knowledge. Its purpose is to exploit an organization's intellectual assets for greater productivity, new value, and increased competitiveness. Owing to globalization and the Internet's impact, many organizations are increasingly geographically dispersed and organized around virtual teams. With the large number of online documents, several document management systems have entered the market. However, these systems have weaknesses:

- *Searching information*: Existing keyword-based searches retrieve irrelevant information that uses a certain word in a different context; they might miss information when different words about the desired content are used.
- *Extracting information*: Current human browsing and reading requires extracting relevant information from information sources. Automatic agents lack the commonsense knowledge required to extract such information from textual representations, and they fail to integrate information spread over different sources.
- *Maintaining*: Sustaining weakly structured text sources is difficult and time-consuming when such sources become large. Keeping such collections consistent, correct, and up to date requires a mechanized representation of semantics and constraints that can help detect anomalies.
- *Automatic document generation*: Adaptive Web sites that enable dynamic reconfiguration according to user profiles or other relevant aspects could prove very useful. The generation of semistructured information presentations from semistructured data requires a machine-accessible representation of the semantics of these information sources.

Using ontologies, semantic annotations will allow structural and semantic definitions of documents. These annotations could provide completely new possibilities: intelligent search instead of keyword matching, query answering instead of information retrieval, document exchange between departments through ontology mappings, and definitions of views on documents.

Web commerce

E-commerce is an important and growing business area for two reasons. First, e-commerce extends existing business models—it reduces costs, extends existing distribution channels, and might even introduce new distribution possibilities. Second, it enables completely new business models and gives them a much greater importance than they had before. What has up to now been a peripheral aspect of a business field can suddenly receive its own important revenue flow.

Examples of business field extensions are online stores; examples of new business fields are shopping agents and online marketplaces and auction houses that turn comparison shopping into a business with its own significant revenue flow. The advantages of online stores and their success stories have led to a large number of shopping pages. The new task for customers is to find a shop that sells the product they're seeking, getting it in the desired quality, quantity, and time, and paying as little as possible for it. Achieving these goals through browsing requires significant time and only covers a small share of the actual offers. Shopbots visit several stores, extract product information, and present it to the customer as an instant market overview. Their functionality is provided through *wrappers*, which use keyword search to find product information together with assumptions on regularities in presentation format and text extraction heuristics. This technology has two severe limitations:

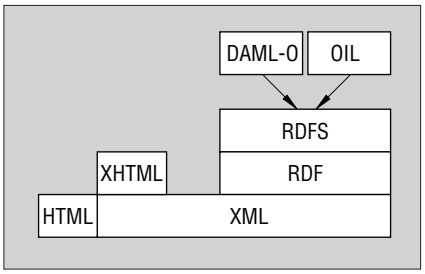


Figure 1. The layer language model for the Web.

- *Effort*: Writing a wrapper for each online store is time-consuming, and changes in store presentation or organization increase maintenance.
- *Quality*: The extracted product information is limited (it contains mostly price information), error-prone, and incomplete. For example, a wrapper might extract the product price, but it usually misses indirect costs such as shipping.

Most product information is provided in natural language; automatic text recognition is still a research area with significant unsolved problems. However, the situation will drastically change in the near future when standard representation formalisms for data structure and semantics are available. Software agents will then *understand* product information. Meta online stores will grow with little effort, which will enable complete market transparency in the various dimensions of the diverse product properties. Ontology mappings, which translate different product descriptions, will replace the low-level programming of wrappers, which is based on text extraction and format heuristics. An ontology will describe the various products and help navigate and search automatically for the required information.

Electronic business

E-commerce in the business-to-business field (B2B) is not new—initiatives to support it in business processes between different companies existed in the 1960s. To exchange business transactions electronically, sender and receiver must agree on a

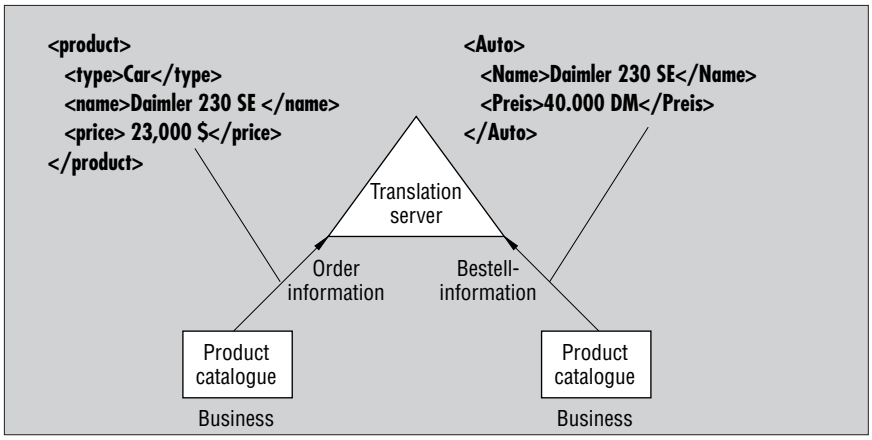


Figure 2. The translation of structure, semantics, and language.

standard (a protocol for transmitting content and a language for describing content). A number of standards arose for this purpose—one of them is the UN initiative, Electronic Data Interchange for Administration, Commerce, and Transport (Edifact). In general, the automation of business transactions has not lived up to the propagandists' expectations, partly because of the serious shortcomings of approaches such as Edifact: It is a procedural and cumbersome standard, making the programming of business transactions expensive and error-prone, and it results in large maintenance efforts. Moreover, the exchange of business data over extranets is not integrated with other document exchange processes—Edifact is an isolated standard.

Using the Internet's infrastructure for business exchange will significantly improve this situation. Standard browsers can render business transactions and transparently integrate them into other document exchange processes in intranet and Internet environments. However, the fact that HTML does not provide a means for presenting rich syntax and data semantics hampers this. XML, which is designed to close this gap in current Internet technology, drastically changes the situation. We can model B2B communication and data exchange with the same means available for other data exchange processes, we can render transaction specifications on standard browsers, and maintenance is cheap. However, although XML provides a standard serialized syntax for defining data structure and semantics, it does not provide standard data structures and terminologies to describe business processes and exchanged products. Therefore, XML-based e-commerce will need ontologies in two important ways:

- *Standard ontologies* must cover the various business areas. In addition to official standards, vertical marketplaces (Internet portals) could generate de facto standards—if they can attract significant shares of a business field's online transactions. Examples include Dublin Core, Common Business Library (CBL), Commerce XML (cXML), ecl@ss, Open Applications Group Integration Specification (OAGIS), Open Catalog Format (OCF), Open Financial Exchange (OFX), Real Estate Transaction Markup Language (RETML), RosettaNet, UN/SPSC (see www.diffuse.org), and UCEC.
- *Ontology-based translation services* must link different data structures in areas where standard ontologies do not exist or where a particular client needs a translation from his or her terminology into the standard. This translation service must cover structural and semantic as well as language differences (see Figure 2).

Ontology-based trading will significantly extend the degree to which data exchange is automated and will create completely new business models in participating market segments.

Why OIL?

Effective, efficient work with ontologies requires support from advanced tools. We need an advanced ontology language to express and represent ontologies. This language must meet three requirements:

- It must be highly intuitive to the human user. Given the success of the frame-based and object-oriented modeling paradigm, an ontology should have a frame-like look and feel.
- It must have a well-defined formal semantics with established reasoning properties to ensure completeness, correctness, and efficiency.
- It must have a proper link with existing Web languages such as XML and RDF to ensure interoperability.

Many of the existing languages such as CycL,[6] KIF,[7] and Ontolingua[8] fail. However, OIL[9] matches these criteria and unifies the three important aspects that different communities provide: epistemologically rich modeling primitives as provided by the frame community, formal semantics and efficient reasoning support as provided by description logics, and a standard proposal for syntactical exchange notations as provided by the Web community.

Frame-based systems

The central modeling primitives of predicate logic are predicates. Frame-based and object-oriented approaches take a different viewpoint. Their central modeling primitives are classes (or frames) with certain properties called *attributes*. These attributes do not have a global scope but apply only to the classes for which they are defined—we can associate the "same" attribute (the same attribute name) with different range and value restrictions when defined for different classes. A frame provides a context for modeling one aspect of a domain. Researchers have developed many other additional refinements of these modeling constructs, which have led to this modeling paradigm's incredible success.

Many frame-based systems and languages have emerged, and, renamed as object orientation, they have conquered the software engineering community. OIL incorporates the *essential modeling primitives* of frame-based systems—it is based on the notion of a *concept* and the definition of its superclasses and attributes. Relations can also be defined not as an attribute of a class but as an independent entity having a certain domain and range. Like classes, relations can fall into a hierarchy.

Description logics

DL describes knowledge in terms of concepts and role restrictions that can automatically derive classification taxonomies. Knowledge representation research's main

thrust is to provide theories and systems for expressing structured knowledge and for accessing and reasoning with it in a principled way. In spite of the discouraging theoretical complexity of the results, there are now efficient implementations for DL languages, which we explain later. OIL inherits from DL its formal semantics and the efficient reasoning support.

Web standards: XML and RDF

Modeling primitives and their semantics are one aspect of an ontology language, but we still have to decide about its syntax. Given the Web's current dominance and importance, we must formulate a syntax of an ontology exchange language with existing Web standards for information representation. First, OIL has a well-defined syntax in XML based on a document type definition and an XML Schema definition. Second, OIL is an extension of RDF and RDFS. With regard to ontologies, RDFS provides two important contributions: a standardized syntax for writing ontologies and a standard set of modeling primitives such as **instance-of** and **subclass-of** relationships.

OIL's layered architecture

A single ontology language is unlikely to fulfill all the needs of the Semantic Web's large range of users and applications. We therefore organized OIL as a series of ever-increasing layers of sublanguages. Each additional layer adds functionality and complexity to the previous one. Agents (humans or machines) that can only process a lower layer can still partially understand ontologies expressed in any of the higher layers. A first and very important application of this principle is the relation between OIL and RDFS. As Figure 3 shows, core OIL coincides largely with RDFS (with the exception of RDFS's reification features). This means that even simple RDFS agents can process OIL ontologies and pick up as much of their meaning as possible with their limited capabilities.

Standard OIL aims to capture the necessary mainstream modeling primitives that provide adequate expressive power and are well understood, thus precisely specifying the semantics and making complete inference viable.

Instance OIL includes a thorough individual integration. Although the previous layer—Standard OIL—includes modeling constructs that specify individual fillers in term definitions, Instance OIL includes a full-fledged database capability.

Heavy OIL will include additional representational (and reasoning) capabilities. A more expressive rule language and metaclass facilities seem highly desirable. We will define these extensions of OIL in cooperation with the DAML (DARPA Agent Markup Language; www.daml.org) initiative for a rule language for the Web.

OIL's layered architecture has three advantages:

- An application is not forced to work with a language that offers significantly more expressiveness and complexity than is needed.
- Applications that can only process a lower level of complexity can still catch some of an ontology's aspects.
- An application that is aware of a higher level of complexity can still understand ontologies expressed in a simpler ontology language.

Defining an ontology language as an extension of RDFS means that every RDFS ontology is a valid ontology in the new language (an OIL processor will also understand RDFS). However, the other direction is also possible: Defining an OIL extension as closely as possible to RDFS allows maximal reuse of existing RDFS-based applications and tools. However, because the ontology language usually contains new aspects (and therefore a new vocabulary, which an RDFS processor does not know), 100 percent compatibility is impossible. Let's look at an example. The following OIL expression defines **herbivore** as a class, which is a subclass of **animal** and disjunct to all **carnivores**:

```
<rdfs:Class rdf:ID="herbivore">
  <rdf:type
      rdf:resource="http://www.
        ontoknowledge.org/oil/RDFS-
          schema/#DefinedClass"/>
  <rdfs:subClassOf rdf:resource="#animal"/>
  <rdfs:subClassOf>
      <oil:NOT>
          <oil:hasOperand rdf:resource="
            #carnivore"/>
      </oil:NOT>
  </rdfs:subClassOf>
</rdfs:Class>
```

An application limited to pure RDFS can still capture some aspects of this definition:

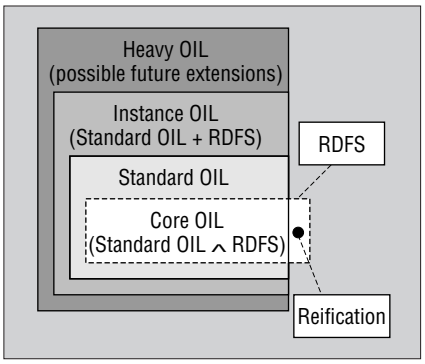


Figure 3. OIL's layered language model.

```
<rdfs:Class rdf:ID="herbivore">
  <rdfs:subClassOf rdf:resource="#animal"/>
  <rdfs:subClassOf>
                ...
  </rdfs:subClassOf>
</rdfs:Class>
```

It encounters that **herbivore** is a subclass of **animal** and a subclass of a second class, which it cannot understand properly. This seems to preserve complicated semantics for simpler applications.

An illustration of the OIL modeling primitive

An OIL ontology is itself annotated with metadata, starting with such things as title, creator, creation date, and so on. OIL follows the W3C Dublin Core Standard on bibliographical meta date for this purpose.

Any ontology language's core is its hierarchy of class declarations, stating, for example, that DeskJet printers are a subclass of printers. We can declare classes as *defined*, which indicates that the stated properties are not only necessary but also sufficient conditions for class membership. Instead of using single types in expressions, we can combine classes in logical expressions indicating intersection, union, and complement of classes.

We can declare *slots* (relations between classes) together with logical axioms, stating whether they are functional (having at most one value), transitive, or symmetric, and stating which (if any) slots are inverse. We can state range restrictions as part of a slot declaration as well as the number of distinct values that a slot may have. We can further restrict slots by *value-type* or *has-value* restrictions. A value-type restriction demands that every value of the property must be of the stated

```
class-def Product
slot-def Price
  domain Product
slot-def ManufacturedBy
  domain Product
class-def PrintingAndDigitalImagingProduct
  subclass-of Product
class-def HPProduct
  subclass-of Product
  slot-constraint ManufacturedBy
    has-value "Hewlett Packard"
class-def Printer
  subclass-of PrintingAndDigitalImagingProduct
slot-def PrinterTechnology
  domain Printer
slot-def Printing Speed
  domain Printer
slot-def PrintingResolution
  domain Printer
class-def PrinterForPersonalUse
  subclass-of Printer
class-def HPPrinter
  subclass-of HPProduct and Printer
class-def LaserJetPrinter
  subclass-of Printer
  slot-constraint PrintingTechnology
    has-value "Laser Jet"
class-def HPLaserJetPrinter
  subclass-of LaserJetPrinter and HPProduct
class-def HPLaserJet1100Series
  subclass-of HPLaserJetPrinter and PrinterFor
    PersonalUse
  slot-constraint PrintingSpeed
    has-value "8 ppm"
  slot-constraint PrintingResolution
    has-value "600 dpi"
class-def HPLaserJet1100se
  subclass-of HPLaserJet1100Series
  slot-constraint Price
    has-value "$479"
class-def HPLaserJet1100xi
  subclass-of HPLaserJet1100Series
  slot-constraint Price
    has-value "$399"
```

Figure 4. A small printer ontology in OIL.

type; has-value restrictions require the slot to have at least values from the stated type.

A crucial aspect of OIL is its formal semantics.[10] An OIL ontology is given a formal semantics by mapping each class into a set of objects and each slot into a set of pairs of objects. This mapping must obey the constraints specified by the definitions of the classes and slots. We omit the details of this formal semantics, but it must exist and be consulted whenever necessary to resolve disputes about the meaning of language constructions. It is an ultimate reference point for OIL applications.

Figure 4 shows a very simple example of an OIL ontology provided by SemanticEdge (www.interprice.com). It illustrates OIL's most basic constructs.

This defines a number of classes and organizes them in a class hierarchy (for example, **HPProduct** is a subclass of **Product**). Various properties (or slots) are defined, together with the classes to which they apply (such as a **Price** is a property of any **Product**, but a **PrintingResolution** can only be stated for a **Printer**, an indirect subclass of **Product**). For certain classes, these properties have restricted values (for example, the price of any HPLaserJet1100se is restricted to $479). In OIL, we can also combine classes by using logical expressions—for example, an **HPPrinter** is both an **HPProduct** and a **Printer** (and consequently inherits the properties from both classes).

OIL tools

OIL has strong tool support in three areas:

- *ontology editors*, to build new ontologies;
- *ontology-based annotation tools*, to link unstructured and semistructured information sources with ontologies; and
- *reasoning with ontologies*, which enables advanced query-answering services, supports ontology creation, and helps map between different ontologies.

Ontology editors

Ontology editors help human knowledge engineers build ontologies—they support the definition of concept hierarchies, the definition attributes for concepts, and the definition of axioms and constraints. They must provide graphical interfaces and conform to existing standards in Web-based software development. They enable the inspecting, browsing, codifying, and modifying of ontologies, and they support ontology development and maintenance tasks. Currently, two editors for OIL are available, and a third is under development:

- *OntoEdit* (see Figure 5) is an ontology-engineering environment developed at the Knowledge Management Group of the University of Karlsruhe, Institute AIFB (http://ontoserver.aifb.uni-karlsruhe.de/ontoedit). Currently, OntoEdit supports Frame-Logic, OIL, RDFS, and XML. It is commercialized from Ontoprise (www.ontoprise.de).
- *OILed* is a freely available and customized editor for OIL implemented by the University of Manchester and sponsored by the Vrije Universiteit, Amsterdam, and SemanticEdge (see http://img.cs.man.ac.uk/oil). OILed aims to provide a simple freeware editor that demonstrates—and stimulates interest in—OIL. OILed is not intended to be a full ontology development environment—it will not actively support the development of large-scale ontologies, the migration and integration of ontologies, versioning, argumentation, and many other activities that are involved in ontology construction. Rather, it is a NotePad for ontology editors that offers just enough functionality to let users build ontologies and demonstrate how to check them for consistency.
- *Protégé*[11] lets domain experts build knowledge-based systems by creating and modifying reusable ontologies and problem-solving methods (see www.smi.stanford.edu/projects/protege). Protégé generates domain-specific knowledge acquisition tools and applications from ontologies. More than 30 countries have used it. It is an ontology editor that can define classes and class hierarchy, slots and slot-value restrictions, and relationships between classes and properties of these relationships. The instances tab is a knowledge acquisition tool that can acquire instances of the classes defined in the ontology. Protégé, built at Stanford University, currently supports RDF—work on extending it to OIL is starting.

Ontology-based annotation tools

Ontologies can describe large instance populations. In OIL's case, two tools currently aid such a process. First, we can derive an XML DTD and an XML Schema definition from an ontology in OIL. Second, we can derive an RDF and RDFS definition for instances from OIL. Both provide means to express large volumes of semistructured information as instance information in OIL. More details appear elsewhere.[4,12,13]

Reasoning with ontologies: Instance and schema inferences

Inference engines for ontologies can reason about an ontology's instances and

schema definition. For example, they can automatically derive the right position of a new concept in a given concept hierarchy. Such reasoners help build ontologies and use them for advanced information access and navigation. OIL uses the FaCT (Fast Classification of Terminologies, www.cs.man.ac.uk/~horrocks/FaCT) system to provide reasoning support for ontology design, integration, and verification. FaCT is a DL classifier that can provide consistency checking in modal and other similar logics. FaCT's most interesting features are its expressive logic, its optimized tableaux implementation (which has now become the standard for DL systems), and its Corba-based client–server architecture. FaCT's optimizations specifically aim to improve the system's performance when classifying realistic ontologies. This results in performance improvements of several orders of magnitude compared with older DL systems. This performance improvement is often so great that it is impossible to measure precisely because nonoptimized systems are virtually nonterminating with ontologies that FaCT can easily deal with.[14] For example, for a large medical terminology ontology developed in the GALEN project,[15] FaCT can check the consistency of all 2,740 classes and determine the complete class hierarchy in approximately 60 seconds of CPU (450-MHz Pentium III) time. FaCT can be accessed through a Corba interface.

Applications of OIL

Earlier, we sketched three application areas for ontologies: knowledge management, Web commerce, and e-business. Not surprisingly, we find applications of OIL in all three areas. On-To-Knowledge (www.ontoknowledge.org)[16] extends OIL to a full-fledged environment for knowledge management in large intranets and Web sites. Unstructured and semistructured data is automatically annotated, and agent-based user interface techniques and visualization tools help users navigate and query the information space. Here, On-To-Knowledge continues a line of research that began with SHOE[17] and Ontobroker:[18] using ontologies to model and annotate the semantics of information resources in a machine-processable manner. On-To-Knowledge is carrying out three industrial case studies to evaluate the tool environment for ontology-based knowledge management.

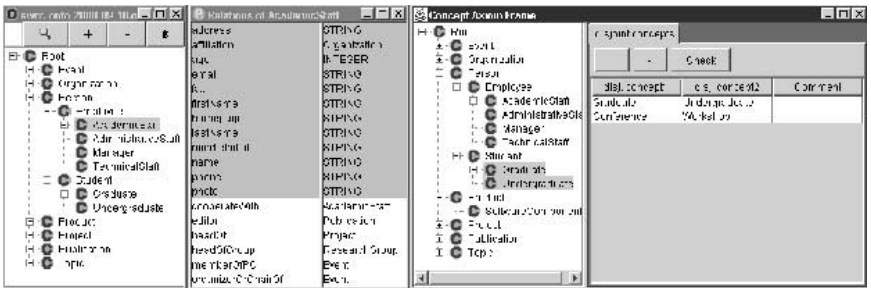


Figure 5. A screen shot of OntoEdit.

Swiss Life: Organizational memory

Swiss Life[19] (www.swisslife.ch) implements an intranet-based front end to an organizational memory with OIL. The starting point is the existing intranet information system, called ZIS, which has considerable drawbacks. Its great flexibility allows for its evolution with actual needs, but this also makes finding certain information difficult. Search engines help only marginally. Clearly, formalized knowledge is connected with weakly structured background knowledge here—experience shows that this is extremely bothersome and error-prone to maintain. The only way out is to apply content-based information access so that we no longer have a mere collection of Web pages but a full-fledged information system that we can rightly call an organizational memory.

British Telecom: Call centers

Call centers are an increasingly important mechanism for customer contact in many industries. What will be required in the future is a new philosophy in customer interaction design. Every transaction should emphasize the uniqueness of both the customer and the customer service person—this requires effective knowledge management (see www.bt.com/innovations), including knowledge about the customer and about the customer service person, so that customers are directed to the correct person in a meaningful and timely way. Some of BT's call centers are targeted to identify opportunities for effective knowledge management. More specifically, call center agents tend to use a variety of electronic sources for information when interacting with customers, including their own specialized systems, customer databases, the organization's intranet, and, perhaps most important, case bases of best practices. OIL provides an intuitive front-end tool to these heterogeneous information sources to ensure smooth transfer to others.

EnerSearch: Virtual enterprise

EnerSearch is a virtual organization researching new IT-based business strategies and customer services in deregulated energy markets (www.enersearch.se).[20] EnerSearch is a knowledge creation company—knowledge that must transfer to its shareholders and other interested parties. Its Web site is one of the mechanisms for this, but finding information on certain topics is difficult—the current search engine supports free-text search rather than content-based search. So, EnerSearch applies the OIL toolkit to enhance knowledge transfer to researchers in the virtual organization in different disciplines and countries and specialists from shareholding companies interested in getting up-to-date R&D information.

OIL has several advantages: it is properly grounded in Web languages such as XML Schemas and RDFS, and it offers different levels of complexity. Its inner layers enable efficient reasoning support based on FaCT, and it has a well-defined formal semantics that is a baseline requirement for the Semantic Web's languages. Regarding its modeling primitives, OIL is not just another new language but reflects certain consensus in areas such as DL and frame-based systems. We could only achieve this by including a large group of scientists in OIL's development. OIL is also a significant source of inspiration for the ontology language

DAML+OIL (www.cs.man.ac.uk/~horrocks/DAML-OIL), developed through the DAML initiative. The next step is to start on a W3C working group on the Semantic Web, taking DAML+OIL as a starting point.

Defining a proper language is an important step to expanding the Semantic Web. Developing new tools, architectures, and applications is the real challenge that will follow. ■

Acknowledgments

We thank Hans Akkermans, Sean Bechhofer, Jeen Broekstra, Stefan Decker, Ying Ding, Michael Erdmann, Carole Goble, Michel Klein, Alexander Mädche, Enrico Motta, Borys Omelayenko, Steffen Staab, Guus Schreiber, Lynn Stein, Heiner Stuckenschmidt, and Rudi Studer, all of whom were involved in OIL's development.

References

1. T. Berners-Lee, *Weaving the Web*, Orion Business Books, London, 1999.
2. O. Lassila and R. Swick, *Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation*, World Wide Web Consortium, Boston, 1999, www.w3.org/TR/REC-rdf-syntax (current 6 Dec. 2000).
3. D. Brickley and R.V. Guha, *Resource Description Framework (RDF) Schema Specification 1.0, W3C Candidate Recommendation*, World Wide Web Consortium, Boston, 2000, www.w3.org/TR/rdf-schema (current 6 Dec. 2000).
4. J. Broekstra et al., "Enabling Knowledge Representation on the Web by Extending RDF Schema," *Proc. 10th Int'l World Wide Web Conf.*, Hong Kong, 2001.
5. T.R. Gruber, "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition*, vol. 5, 1993, pp. 199–220.
6. D.B. Lenat and R.V. Guha, *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project*, Addison-Wesley, Reading, Mass., 1990.
7. M.R. Genesereth, "Knowledge Interchange Format," *Proc. Second Int'l Conf. Principles of Knowledge Representation and Reasoning* (KR 91), J. Allenet et al., eds., Morgan Kaufmann, San Francisco, 1991, pp. 238–249; http://logic.stanford.edu/kif/kif.html (current 9 Mar. 2001).
8. A. Farquhar, R. Fikes, and J. Rice, "The Ontolingua Server: A Tool for Collaborative Ontology Construction," *Int'l. J. Human–Computer Studies*, vol. 46, 1997, pp. 707–728.

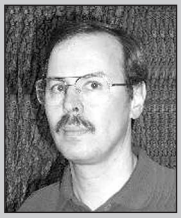
The Authors

Dieter Fensel's biography appears in the Guest Editors' Introduction on page 25.

Ian Horrocks is a lecturer in computer science with the Information Management Group at the University of Manchester, UK. His research interests include knowledge representation, automated reasoning, optimizing reasoning systems, and ontological engineering, with particular emphasis on the application of these techniques to the World Wide Web. He received a PhD in computer science from the University of Manchester. He is a member of the OIL language steering committee and the Joint EU/US Committee on Agent Markup Languages, and is coeditor of the DAML+OIL language specification. Contact him at the Dept. of Computer Science, Univ. of Manchester, Oxford Rd., Manchester, M13 9PL, UK; horrocks@cs.man.ac; www.cs.man.ac.uk/~horrocks.

Frank van Harmelen is a senior lecturer in the AI Department at Vrije Universiteit in Amsterdam. His research interests include specification languages for knowledge-based systems, using languages for validation and verification of KBS, developing gradual notions of correctness for KBS, and verifying weakly structured data. He received a PhD in artificial intelligence from the University of Edinburgh. Contact him at Dept. of AI, Faculty of Sciences, Vrije Universiteit Amsterdam de Boelelaan 1081a, 1081HV Amsterdam, Netherlands; frank.van.harmelen@cs.vu.nl; www.cs.vu.nl/~frankh.

Deborah L. McGuinness is the associate director and senior research scientist for the Knowledge Systems Laboratory at Stanford University. Her main research areas include ontologies, description logics, reasoning systems, environments for building and maintaining information, knowledge-enhanced search, configuration, and intelligent commerce applications. She also runs a consulting business dealing with ontologies and artificial intelligence for business applications and serves on several technology advisory boards and academic advisory boards. She received a PhD from Rutgers University in reasoning systems. Contact her at the Knowledge Systems Laboratory, Stanford Univ., Stanford, CA 94305; dlm@ksl.stanford.edu; www.ksl.stanford.edu/people/dm.



Peter F. Patel-Schneider is a member of the technical staff at Bell Labs Research. His research interests center on the properties and use of description logics. He is also interested in rule-based systems, including standard systems derived from OPS as well as newer formalisms such as R++. He received his PhD from the University of Toronto. Contact him at Bell Labs Research, 600 Mountain Ave., Murray Hill, NJ 07974; pfps@research.bell-labs.com; www.bell-labs.com/user/pfps.

9. D. Fensel et al., "OIL in a Nutshell," *Proc. European Knowledge Acquisition Conference* (EKAW 2000), R. Dieng et al., eds., *Lecture Notes in Artificial Intelligence*, no. 1937, Springer-Verlag, Berlin, 2000, pp. 1–16.

10. I. Horrocks et al., *The Ontology Inference Layer OIL*, tech. report, Vrije Universiteit Amsterdam; www.ontoknowledge.org/oil/TR/oil.long.html (current 9 Mar. 2001).

11. W.E. Grosso et al., "Knowledge Modeling at the Millennium (The Design and Evolution of Protege-2000)," *Proc. 12th Workshop Knowledge Acquisition, Modeling, and Management*, 1999.

12. M. Klein et al., "The Relation between Ontologies and Schema-Languages: Translating OIL Specifications to XML Schema," *Proc. Workshop on Applications of Ontologies and Problem-Solving Methods, 14th European Conf. on Artificial Intelligence*, Berlin, 2000.

13. M. Erdmann and R. Studer, "How to Structure and Access XML Documents with Ontologies," *Data and Knowledge Eng.*, vol. 36, no. 3, 2001.

14. I. Horrocks and P.F. Patel-Schneider, "Optimizing Description Logic Subsumption," *J. Logic and Computation*, vol. 9, no. 3, June 1999, pp. 267–293.

15. A.L. Rector, W.A. Nowlan, and A. Glowinski, "Goals for Concept Representation in the GALEN Project," *Proc. 17th Ann. Symp. Computer Applications in Medical Care* (SCAMC 93), McGraw-Hill, New York, 1993, pp. 414–418.

16. D. Fensel et al., "On-To-Knowledge: Ontology-based Tools for Knowledge Management," *Proc. eBusiness and eWork 2000 Conf.* (EMMSEC 2000), 2000.

17. S. Luke, L. Spector, and D. Rager, "Ontology-Based Knowledge Discovery on the World Wide Web," *Proc. 13th Nat'l Conf. Artificial Intelligence* (AAAI 96), American Association for Artificial Intelligence, Menlo Park, Calif., 1996.

18. D. Fensel et al., "Lessons Learned from Applying AI to the Web," *J. Cooperative Information Systems*, vol. 9, no. 4, Dec. 2000, pp. 361–382.

19. U. Reimer et al., eds., *Proc. Second Int'l Conf. Practical Aspects of Knowledge Management* (PAKM 98), 1998.

20. F. Ygge and J.M. Akkermans, "Decentralized Markets versus Central Control: A Comparative Study," *J. Artificial Intelligence Research*, vol. 11, July–Dec. 1999, pp. 301–333.

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.

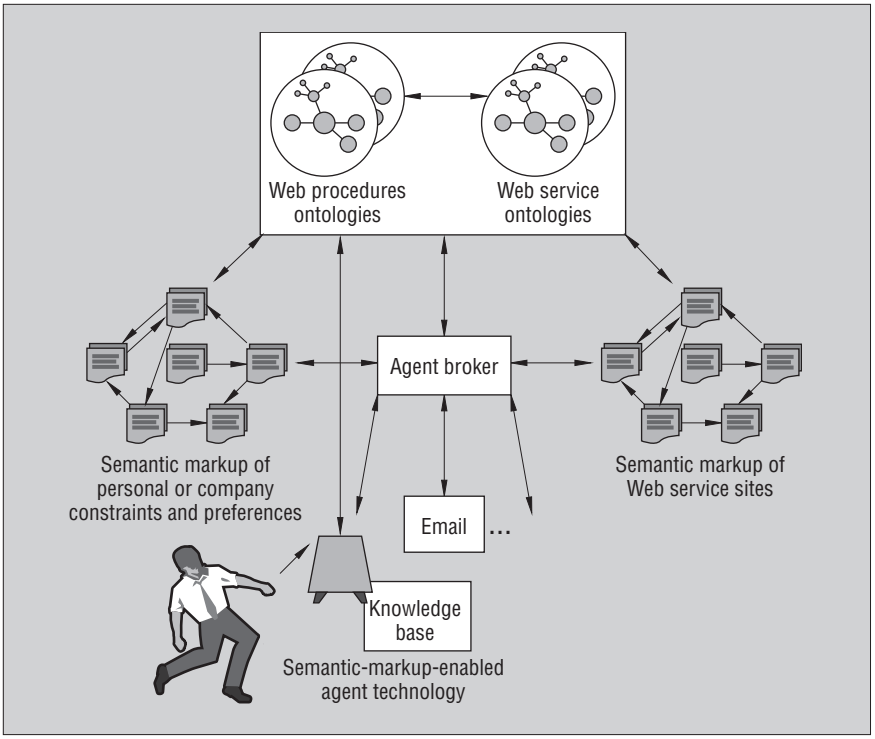


Figure 1. A framework for Semantic Web services.

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)

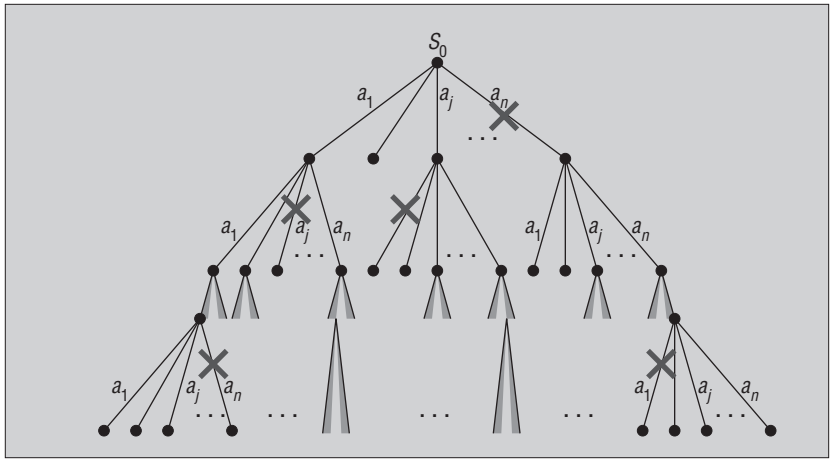


Figure 2. The tree of situations.

```
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.

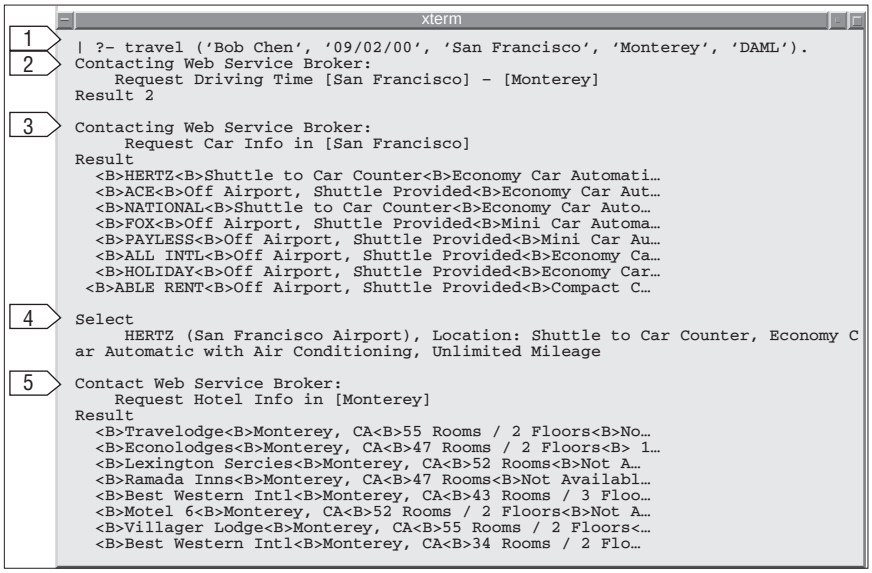


Figure 4. Agent interacting with Web services through OAA.

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.

A Portrait of the Semantic Web in Action

Jeff Heflin and James Hendler, *University of Maryland*

Without semantically enriched content, the Web cannot reach its full potential. The authors discuss tools and techniques for generating and processing such content, thus setting a foundation upon which to build the Semantic Web.

The Web's phenomenal growth rate makes it increasingly difficult to locate, organize, and integrate the available information. To cope with the enormous quantity of data, we need to hand off portions of these tasks to machines. However, because natural-language processing is still an unsolved problem, machines cannot understand the Web pages to the extent required to perform the desired tasks.

An alternative is to change the Web to make it more understandable by machines, thereby creating a Semantic Web. Many researchers believe the key to building this new Web lies in the development of semantically enriched languages. Early languages, such as the resource description framework,[1] Simple HTML Ontology Extensions (SHOE),[2] and Ontobroker,[3] have led to more recent efforts, such as the Defense Advanced Research Projects Agency's Agent Markup Language (DAML). Some say that languages such as these will revolutionize the Web. If so, how will the new Web work?

In this article, we put a Semantic Web language through its paces and try to answer questions about how people can use it, such as:

- How do authors generate semantic descriptions?
- How do agents discover these descriptions?
- How can agents integrate information from different sites?
- How can users query the Semantic Web?

We present a system that addresses these questions and describe tools that help users interact with the Semantic Web. We motivate the design of our system with a specific application: semantic markup in the computer science domain.

Producing semantic markup

Describing a set of Web pages using a Semantic Web language can be challenging. (For an overview of Semantic Web languages, see the related sidebar.) The first step is to consider the pages' domain and choose an appropriate ontology. As Semantic Web languages evolve, knowledge engineers will likely provide huge ontology libraries, as well as numerous search mechanisms to help users find relevant ontologies. Meanwhile, some of the common languages provide starter ontology libraries. (Knowledge engineering, which covers the difficult process of designing ontologies, is outside this article's scope.)

Our running example uses the SHOE language, which has served as a testbed for Semantic Web ideas over the past five years, although technically the discussion could apply to any Semantic Web language. SHOE has a computer science department ontology that includes classes such as Student, Faculty, Course, Department, Publication, and Research, and relations such as publicationAuthor, member, emailAddress, and advisor. This ontology's scope makes it relevant to faculty and student homepages, department Web pages, research project Web pages, and publication indices. Authors can use a number of methods to produce SHOE markup for these pages.

Authoring tools

As with HTML, authors can use a text editor to

1094-7167/01/$10.00 © 2001 IEEE

add semantic markup to a page. However, unlike HTML processors, Semantic Web processors are not very forgiving, and errors can result in the processors ignoring large portions of the annotations. One solution is to provide authoring tools that let authors create markup by making selections and filling in forms. For the SHOE project, we developed the Knowledge Annotator (see Figure 1) to perform this function.

In SHOE, a document references a set of ontologies that provide the vocabulary used to describe entities (called *instances*). Each assertion about an instance is called a *claim*, to denote that it may not necessarily be true. The Knowledge Annotator has an interface that displays instances, ontologies, and claims, and a user can add, edit, or remove any of these objects. When creating a new object, the Knowledge Annotator prompts the user for the necessary information. In the case of claims, the user can choose the source ontology from a list and then choose categories or relations defined in that ontology from another list. The available relations are automatically filtered based on whether the instances entered can fill the argument positions.

Users have access to various methods for viewing the knowledge in the document. These include a view of the source HTML, a logical notation view, and a view that organizes claims by subject and describes them using simple English. In addition to prompting the user for inputs, the tool performs error checking to ensure correctness and converts the inputs into legal SHOE syntax. For these reasons, only a rudimentary understanding of SHOE is necessary to mark up Web pages. If developers enhance contemporary Web authoring tools with semantic markup authoring capabilities, adding semantic markup could become a regular activity in the Web-page design process.

Members of our research group provided markup for their homepages and those of the

Overview of Semantic Web languages

Unlike Extensible Markup Language (XML), which uses a name or prose description to imply meaning in documents, a Semantic Web language must describe meaning in a machine-readable way. Therefore, the language needs not only the ability to specify a vocabulary, but also to formally define the vocabulary so that it will work in automated reasoning. As such, the subfield of AI known as knowledge representation greatly influences Semantic Web languages.

However, to meet the needs of the Web, Semantic Web languages must also differ from traditional KR languages. The most obvious difference is syntactical: Language designers base Semantic Web syntaxes on existing standards such as Hypertext Markup Language (HTML) or XML so that integration with other Web technologies is possible. Other differences depend on the nature of the Web.

Because the Web is decentralized, the language must allow for the definition of diverse—and potentially conflicting—vocabularies. To handle the Web's rapid evolution, the language must let the vocabularies evolve as human understanding of their use improves. Finally, the Web's size requires that scalability play a role in any solution.

An author can formally specify a Semantic Web vocabulary using an ontology or a schema. Such ontologies and schemas are also typically sharable (so users can agree to use the same definitions) and extensible (so users can agree on some common set of definitions but add terms and definitions as necessary). Researchers expect that ontology hierarchies will develop, with top-level abstract ontologies at the root and domain-specific ontologies at the leaves. Thus, automatic interoperability between a pair of ontologies exists to the degree that they share a common ancestor. The language's expressivity determines the potential richness of an ontology's definitions. Most languages let ontologies define class taxonomies so that it is possible to say, for example, a car is a type of vehicle. They also allow for the definition of properties for each class and relationships between multiple classes. Some languages might also allow the formation of more complex definitions, using axioms from some form of logic.

Major differences exist between the leading Semantic Web languages. The resource description framework (RDF) Schema[1] is the least expressive. It is based on a semantic network model, with special links for defining category and property taxonomies and links for applying domain and range constraints to properties. Simple HTML Ontology Extensions (SHOE)[2] is based on a frame system but also allows Horn clause axioms (essentially, simple if–then rules), which authors can use to define things not possible in RDF. More so than its peers, SHOE focuses on dealing with the problems of a dynamic, distributed environment.[3] The Ontology Inference Layer (OIL), based on a frame system augmented with description logic, lets authors express different kinds of definitions.[4] The Defense Advanced Research Projects Agency Agent Markup Language (DAML) is a language under development with the intent to combine the best features of RDF, SHOE, and OIL.

Although ontologies are crucial to making a Semantic Web language work, they merely serve to standardize and provide interpretations for Web content. To make this content machine understandable, the Web pages must contain semantic markup—that is, descriptions which use the terminology that one or more ontologies define. The semantic markup might state that a particular entity is a member of a class, an entity has a particular property, or two entities have some relationship between them. By committing to an ontology, the semantic markup sanctions inferences based on the ontology definitions and lets agents conclude things that the markup implies.

References

1. O. Lassila, "Web Metadata: A Matter of Semantics," *IEEE Internet Computing*, vol. 2, no. 4, July 1998, pp. 30–37.
2. S. Luke et al., "Ontology-Based Web Agents," *Proc. First Int'l Conf. Autonomous Agents*, ACM Press, New York, 1997.
3. J. Heflin and J. Hendler, "Dynamic Ontologies on the Web," *Proc. 17th Nat'l Conf. AI (AAAI-2000)*, MIT–AAAI Press, Menlo Park, Calif., 2000, pp. 443–449.
4. S. Decker et al., "Knowledge Representation on the Web," *Proc. 2000 Int'l Workshop on Description Logics (DL2000)*, Sun SITE Central Europe (CEUR), Aachen, Germany, 2000, http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-33/.

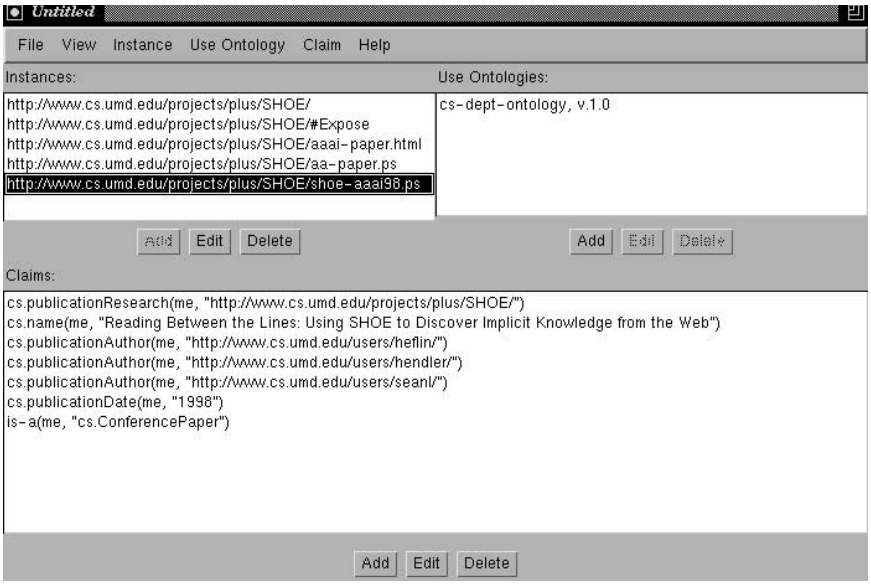


Figure 1. The Knowledge Annotator. Here, the interface is being used to view semantic markup about a Simple HTML Ontology Extensions (SHOE) publication.

group's Web site. Most used the Knowledge Annotator, but some preferred a text editor. Although we produced detailed markup for a set of pages, the set is too small to be of use for anything but controlled demos.

Generating markup on a large scale

For semantic markup to be really useful, it needs to be ubiquitous, but using an authoring tool to generate a lot of markup is tedious. Detractors of the Semantic Web language approach often cite the difficulty of producing markup as the main reason it won't work. Fortunately, there are automatic and semiautomatic approaches for generating semantic markup.

Running SHOE. Some Web pages have regular structure, with labeled fields, lists, and tables. Often, an analyst can map these structures to an ontology and write a program to translate portions of the Web page into the semantic markup language. We developed Running SHOE (see Figure 2), a tool that helps users specify how to extract SHOE markup from these kinds of Web pages. The user selects a page to markup and creates a wrapper for it by specifying a series of delimiters that describe how to extract interesting information. These delimiters indicate the start of a list (so that the program can skip header information) and end of a list (so that it can ignore trailing information); the start and end of a record; and for each field of interest, a pair of start and end delimiters.

A fundamental problem in distributed systems is knowing when markup from different people describes the same entity. If we are to integrate descriptions about such an entity, we must use a common identifier (or key) when referring to it. A URL can often serve as this key because it identifies exactly one Web page, which a single person or organization owns. The regular pages that work best with Running SHOE tend to have lists of things, and each item in each list typically contains a hyperlink to a thing's homepage. However, these hyperlinks often use relative URLs, which are not unique. To handle this problem, the user can specify that a particular field is a URL, so that when the program extracts the data, it expands all relative URLs using the page's URL as a base.

After the user has specified the delimiters, the tool can display a table with a row for each record and a column for each field. Irregularities in a page's HTML code can often cause the program to extract fields or records improperly; this table lets the user verify the results before proceeding. The next step is to convert the table into SHOE markup. In the top-right panel, the user can specify ontology information and a series of templates for SHOE classification and relation declarations.

For each classification or relation argument, the user can specify a literal value or reference a field. At the user's command, the tool can then iterate through these templates and the table of records to create a series of SHOE statements. Using this tool, a trained user can extract substantial markup from a Web page in minutes. Furthermore, because Running SHOE lets users save and retrieve templates, it is easy to regenerate new SHOE markup if the page's content changes.

Computer science department Web sites often have faculty, project, course, and user lists that have ideal formats for Running SHOE. Each item in each list contains an <A> tag that provides the URL of the item's homepage, and this element's content is often the name of the entity being linked to, providing us with a value for the "name" relation. Other properties of the instance often follow and are delimited by punctuation, emphasis, or special spacing. With this tool, a single user can create SHOE markup about the faculty, courses, and projects of 15 major computer science departments in a day.

Although there are many pages for which Running SHOE is useful, there are other important resources from which it cannot extract information. An example of such a site is CiteSeer (http://citeseer.nj.nec.com/cs), an index of online computer science publications that we wanted to integrate with our department Web pages. Interaction with CiteSeer involves issuing a query to one page, viewing a results page, and then selecting a result to get a page about a particular publication. This multistep process prevents Running SHOE from extracting markup from the CiteSeer site.

Publication SHOE Maker. To extract SHOE from CiteSeer, we built a tool called Publication SHOE Maker. PSM issues a query to get publications likely to be from a particular institution and retrieves a fixed number of publication pages from the results. The publication pages contain the publication's title, authors, year, links to online copies, and occasionally additional BibTex information. Each publication page's layout is very similar, so PSM can extract the values of the desired fields easily.

An important issue is how to link the author information with the faculty instances extracted from the department Web pages. Fortunately, CiteSeer includes homepage information, which HomePageSearch (http://hpsearch.uni-trier.de) generates for each author. By using these URLs (as opposed to the authors' names), PSM can establish links to the appropriate instances.

Running SHOE and PSM are only two

examples of tools with which authors can generate markup. Other extraction tools might include machine-learning[4,5] or natural-language-processing techniques. As Extensible Markup Language becomes ubiquitous on the Web, generating wrappers will become easier, and authors will be able to use style sheets to transform a simple XML vocabulary into a semantically enriched one.

If a Web page's provider is willing to include semantic markup, the process can be even easier. For example, databases hold much of the Web's data, and scripts produce Web pages from that data. Because databases are structured resources, an analyst can determine the semantics of a database schema, map the schema to an ontology, and modify the scripts that produce the Web pages to include the appropriate semantic markup.

Integrating resources

After authors have described a number of diverse Web sites with semantic markup, the next problem is determining how to integrate the information. Information integration systems, such as Ariadne,[6] can be useful when developing an application that combines data from a finite number of predetermined sources, but are less helpful when integrating information "on the fly" is necessary. One solution mirrors the operation of contemporary search engines by crawling the Web and storing the information in a central repository.

Exposé

Exposé is a Web crawler that searches for Web pages with SHOE markup and interns the knowledge. A Web crawler essentially performs a graph traversal where the nodes are Web pages and the arcs are the hypertext links between them. When Exposé discovers a new URL, it assigns the page a cost and uses this cost to schedule when it will load that page. Thus, the cost function determines a traversal order. We assume SHOE pages will tend to be localized and interconnected. Therefore, we use a cost function that increases with distance from the start node, where paths through nonSHOE pages are more expensive than those through SHOE pages, and paths that stay within the same directory on the same server are cheaper than those that do not.

Exposé parses each Web page, and if a page references an ontology that Exposé is unfamiliar with, it loads the ontology also. To update its list of pages to visit, Exposé identifies all of the hypertext links, category instances, and relation arguments within the page and evaluates each new URL as we discussed. Finally, the agent stores SHOE category and relation statements and any new ontology information in a knowledge base.

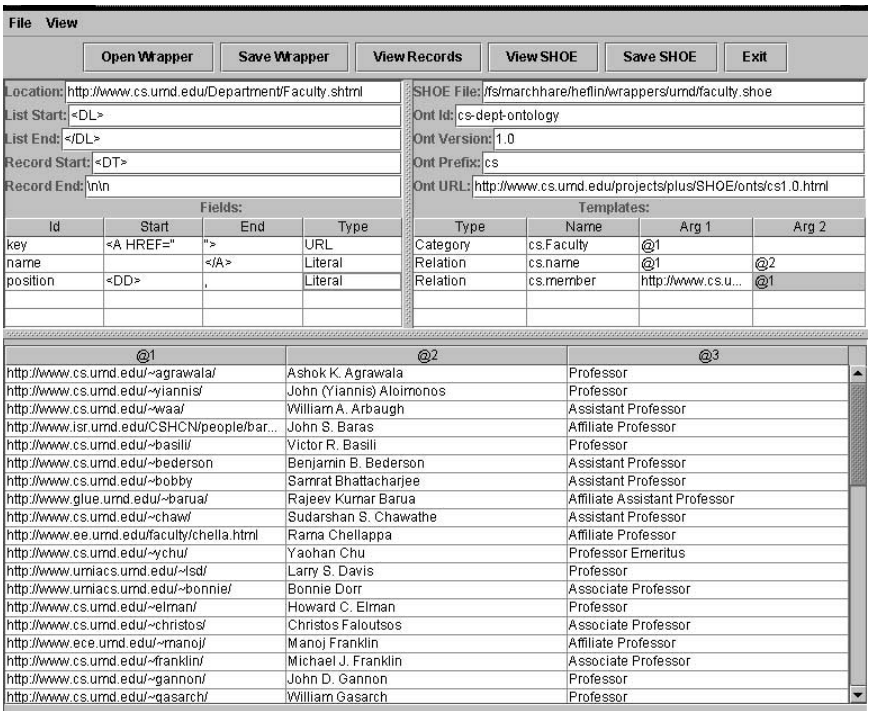


Figure 2. Running SHOE. A user can specify delimiters for recognizing fields and records, verify that they are extracted correctly, then create templates that translate the data into SHOE format.

This KB will determine the system's query capabilities, and thus we must choose an appropriate knowledge representation system. Our SHOE tools all use a generic application programming interface for interaction with the KB, letting us easily use different backends. We have implemented versions of this API for Parka, a high-performance frame system;[7] XSB, a deductive database;[8] and Open Knowledge Base Connectivity-compliant KBs.[9]

By changing the back-end knowledge representation system, we get varying trade-offs between query response time and the degree to which the system uses inference to compute answers. For example, Parka answers recognition queries on large KBs (containing millions of assertions) in seconds, and when used on parallel machines, it provides even better performance. However, Parka's only inferential capabilities are class membership and inheritance, so it cannot use the extra Horn clause rules that SHOE allows. However, XSB can reason with these rules but is not as efficient as Parka. Alternately, the KB could be a relational or object database, providing the greatest scalability and best query response times but sacrificing the ability to infer additional answers. Clearly, the choice of the back-end system depends on the application's expected query needs.

We let Exposé crawl the various computer science Web pages described earlier, and it was able to gather approximately 40,000 assertions. The crawler stored these assertions in both Parka and XSB KBs. Technically we did not need a Web crawler for our example, because we knew the locations of all the relevant pages a priori. However, in an ideal Semantic Web situation, the markup is the product of many individuals working independently, and users could not easily locate it without a crawler.

Querying the Semantic Web

Both general-purpose and domain-specific query tools can access the SHOE knowledge after it has been loaded into the KB. The SHOE Search tool (see Figure 3) is a general-purpose tool that gives users a new way to browse the Web by letting them submit structured queries and open documents by clicking on the URLs in the results. The user first chooses an ontology against which to

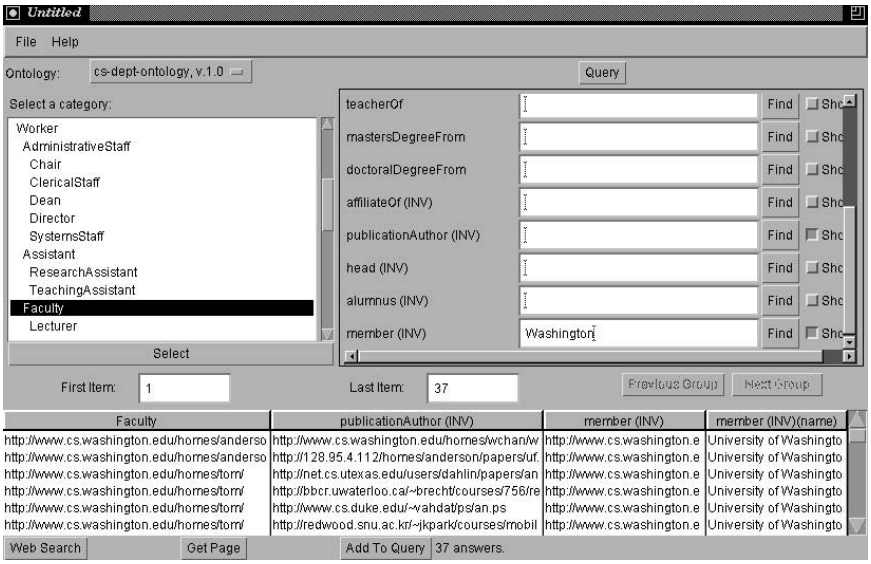


Figure 3. SHOE Search. With this tool, a user issues a query by choosing an ontology, choosing a category from that ontology, and then filling in desired values for properties that instances of that category might have.

issue the query (which essentially establishes a context for the query).

The user then chooses the desired object's class from a hierarchical list and is presented with a list of all properties that could apply to that object. After entering desired values for one or more of these properties, the user issues a query and receives a set of results in a table. If the user double-clicks on a binding that is a URL, the corresponding Web page will open in a new browser window. Thus, the user can browse the Semantic Web.

If SHOE markup does not describe all of the relevant Web pages, SHOE Search's standard query method will not be able to return an answer or might only return partial answers. Therefore, we also have a Web Search feature that translates the user's query into a similar search engine query and submits it to any of a number of popular search engines. Using SHOE Search in this way has two advantages over using the search engines directly:

- By prompting the user for values of properties, it increases the chance that the user will provide distinguishing information for the desired results.
- By automatically creating the query, it can exploit helpful features that users often overlook, such as quoting phrases or using the plus sign to indicate a mandatory term.

We build a query string that comprises a quoted short name for the selected category and, for each property value that the user specifies, a short phrase describing the property. The user's value, which we quote and precede with a plus sign to indicate that it is a mandatory phrase, follows the phrase describing the property.

With SHOE Search, a user can submit many queries pertinent to our computer science domain. Figure 3 shows a sample query to locate University of Washington faculty members and their publications. The computer science ontology serves as a unifying framework for integrating information from the university's faculty page with publication information from CiteSeer. Furthermore, the ontology lets the query system recognize that anyone declared a **Professor** is also **Faculty**.

Sample queries to the KB exposed one problem with the system: Sometimes it didn't integrate information from a department Web page and CiteSeer as expected. These sites occasionally use different URLs to refer to the same person. This is a fundamental problem with using URLs as keys in a Semantic Web system: Multiple URLs can refer to the same Web page because of multiple host names for a given IP address, default pages for a directory URL, host-dependent shortcuts such as a tilde for the users directory, symbolic links within the host, and so on. Additionally, some individuals might have multiple URLs that make equally valid keys for them, such as the URLs of both professional and personal homepages. These problems would be partially alleviated if the language included the ability to specify identifier equivalence—a feature absent from SHOE but present in DAML.

We created a search engine called Semantic Search that is based on the technologies we describe. Semantic Search uses the SHOE Search tool as a query interface and provides utilities for authors, including links to an ontology library, the Knowledge Annotator, an online SHOE validation form, and a form for submitting new pages to the repository. We encourage readers to add markup to their own Web pages and submit them. Semantic Search is available at http://www.cs.umd.edu/projects/plus/SHOE/search/.

We have described a simple architecture for a Semantic Web system that parallels the way contemporary Web tools and search engines work. As Figure 4 shows, authors use various tools to add markup to Web pages, and a Web crawler discovers the information and stores it in a repository, which other tools can then query. Generally, authors need not produce all markup by hand; in many cases, simple extraction tools can generate accurate markup with minimal human effort. Although the tools that comprise this architecture are designed for use with the SHOE language, developers can create similar tools for other Semantic Web languages. Because any number of tools can produce and process the semantic markup on a Web page, other architectures are also possible. For example, developers could create an agent that queries pages directly as opposed to issuing queries to a Web-crawler-constructed repository.

If we achieve the Semantic Web vision, locating useful information on the Internet will be easier, and integrating diverse resources will be simpler. The first step is to design languages that we can use to express explicit semantics. The next step is to improve the systems and tools we describe, so users can naturally provide and receive information on the Semantic Web. Obviously, we must still overcome some obstacles: We need better schemes for ensuring interoperability between independently developed ontologies and approaches for determining who and what to trust. However,

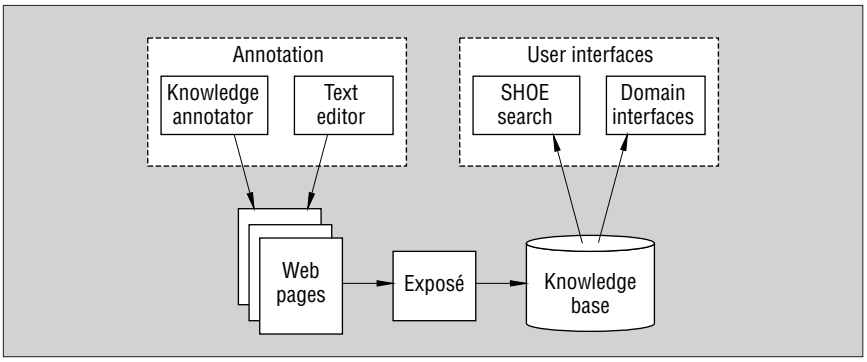


Figure 4. A simple Semantic Web system based on the tools we describe.

these challenges do not appear to be insurmountable, and the Semantic Web could be just around the corner. ■

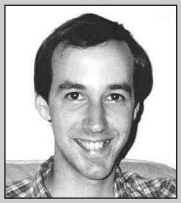
Acknowledgments

The US Air Force Research Laboratory supported this work under grant F306029910013.

References

1. O. Lassila, "Web Metadata: A Matter of Semantics," *IEEE Internet Computing*, vol. 2, no. 4, July 1998, pp. 30–37.
2. S. Luke et al., "Ontology-Based Web Agents," *Proc. First Int'l Conf. Autonomous Agents*, ACM Press, New York, 1997, pp. 59–66.
3. D. Fensel et al., "Ontobroker: How to Enable Intelligent Access to the WWW," *AAAI-98 Workshop on AI and Information Integration*, AAAI Press, Menlo Park, Calif., 1998, pp. 36–42.
4. D. Freitag, "Information Extraction from HTML: Application of a General Machine Learning Approach," *Proc. 15th Nat'l Conf. AI (AAAI-98)*, MIT–AAAI Press, Menlo Park, Calif., 1998, pp. 517–523.
5. N. Kushmerick, D. Weld, and R. Doorenbos, "Wrapper Induction for Information Extraction," *Proc. 15th Int'l Joint Conf. AI*, Morgan Kaufmann, San Francisco, 1997, pp. 729–735.
6. C. Knoblock et al., "Modeling Web Sources for Information Integration," *Proc. 15th Nat'l Conf. AI (AAAI-98)*, MIT–AAAI Press, Menlo Park, Calif., 1998, pp. 211–218.
7. K. Stoffel, M. Taylor, and J. Hendler, "Efficient Management of Very Large Ontologies," *Proc. 14th Nat'l Conf. AI (AAAI-97)*, MIT–AAAI Press, Menlo Park, Calif., 1997.
8. K. Sagonas, T. Swift, and D. Warren, "XSB as an Efficient Deductive Database Engine," *Proc. 1994 ACM SIGMOD Int'l Conf. Management of Data (SIGMOD'94)*, ACM Press, New York, 1994, pp. 442–453.
9. V. Chaudhri et al., "OKBC: A Programmatic Foundation for Knowledge Base Interoperability," *Proc. 15th Nat'l Conf. AI (AAAI-98)*, MIT–AAAI Press, Menlo Park, Calif., 1998, pp. 600–607.

The Authors



Jeff Heflin is a PhD candidate in the Computer Science Department at the University of Maryland. His research interests include Semantic Web languages, ontologies, Internet agents, and knowledge representation. He has worked in the computer consulting industry for four years as a data modeler, database designer, and database administrator. He received a BS in computer science from the College of William and Mary and an MS in computer science from the University of Maryland. He is currently a member of the Joint US–EU Ad hoc Agent Markup Language Committee. Contact him at the University of Maryland, Dept. of Computer Science, College Park, MD 20742; heflin@cs.umd.edu.

James Hendler's biography appears on p. 37.

Creating Semantic Web Contents with Protégé-2000

Natalya F. Noy, Michael Sintek, Stefan Decker, Monica Crubézy, Ray W. Fergerson, and Mark A. Musen, *Stanford University*

As researchers continue to create new languages in the hope of developing a Semantic Web, they still lack consensus on a standard. The authors describe how Protégé-2000—a tool for ontology development and knowledge acquisition—can be adapted for editing models in different Semantic Web languages.

Because we can process only a tiny fraction of information available on the Web, we must turn to machines for help in processing and analyzing its contents. With current technology, machines cannot understand and interpret the meaning of the information in natural-language form, which is how most Web information is represented today. We need a Semantic Web to express information in a precise, machine-interpretable form, so software agents processing the same set of data share an understanding of what the terms describing the data mean.[1]

Consequently, we've recently seen an explosion in the number of Semantic Web languages developed. Because researchers and developers haven't yet reached a consensus on which language is the most suitable, which features each language must have, or which syntax is the most appropriate, we are likely to see even more languages emerge. We need to develop tools that will let us experiment with these new languages so we can compare their expressiveness and features, change language specifications, and select a suitable language for a specific task.

In this article, we describe Protégé-2000, a graphical tool for ontology editing and knowledge acquisition that we can adapt to enable conceptual modeling with new and evolving Semantic Web languages. Protégé-2000 lets us think about domain models at a conceptual level without having to know the syntax of the language ultimately used on the Web. We can concentrate on the concepts and relationships in the domain and the facts about them that we need to express. For example, if we are developing an ontology of wines, food, and appropriate wine–food combinations, we can focus on Bordeaux and lamb instead of markup tags and correct syntax.

Naturally, designing a new tool specifically for a new language could be better than adapting an existing tool. We can offer several reasons, however, for adapting an existing tool at the stage where no single language has emerged as the winner. First, we can experiment with emerging languages without committing enormous amounts of resources to creating tools that are custom-tailored for these languages—only to decide later that the languages are not suitable. Second, Protégé-2000 already provides considerable functionality that a new tool will need to replicate, both at the modeling and user-interface levels. Third, using different customizations of the same tool to edit ontologies in different languages gives us most of the translation among the models in the languages "for free." Translating a model from one language to another becomes as easy as selecting a "save as…" item from a menu.

Semantic Web languages

AI researchers have used ontologies for a long time to express formally a shared understanding of information. An ontology is an explicit specification of the concepts in a domain and the relations among them, which provides a formal vocabulary for information exchange. Specific instances of the concepts defined in the ontologies—*instance data*—paired with ontologies constitute the basis of the Semantic Web. In recent experiments to prototype the Semantic Web, members of different communities with different backgrounds and goals in mind have created a multitude of languages for representing ontologies and instance data on the Web (see Table1). Typically, a Semantic Web language for describing ontologies and instance data contains a hierarchical description

1094-7167/01/$10.00 © 2001 IEEE

Table 1. A selection of Semantic Web languages.

Language	Description	URL
XOL	XML-based ontology-exchange language	www.ai.sri.com/~pkarp/xol
Topic Maps	ISO standard for describing knowledge structures	www.topicmaps.org
SHOE	Simple HTML Ontology Extensions	www.cs.umd.edu/projects/plus/SHOE
RDF and RDFS	Resource description framework and RDF Schema	www.w3.org/RDF
DAML+OIL	DARPA Agent Markup Language + Ontology Inference Layer	www.daml.org

of important concepts in a domain (*classes*). Individuals in the domain are *instances* of these classes, and properties (*slots*) of each class describe various features and attributes of the concept. Logical statements describe relations among concepts. For example, consider an ontology describing wines, food, and appropriate wine–food combinations. Some of the classes describing this domain are **Wine**, **Wineries**, and different types of **Food**. Some properties of the **Wine** class include the wine's **flavor**, **body**, **sugar level**, and the **winery** that produced it.

These notions are present in many Semantic Web languages existing today including SHOE, Topic Maps, XOL, RDF and RDFS, and DAML+OIL.

The SHOE (Simple HTML Ontology Extensions) language, developed at the University of Maryland, introduces primitives to define ontology and instance data on Web pages. Classes are called *categories* in SHOE. Categories constitute a simple *is-a* hierarchy, and slots are binary relations. SHOE also allows relations among instances or instances and data to have any number of arguments (not just binary relations). Horn clauses express intensional definitions in SHOE.

The Hytime community developed Topic Maps, a recent ISO standard (ISO/IEC 13250). Topic Maps aim to annotate documents with conceptual information. *Topics* correspond to classes in other ontology languages and can be linked to documents. Topics are instances of *Topic Types* (other topics), which can be related to one another with *Associations*. Associations correspond closely to slots in other ontology languages. Associations belong to *Association Types*, which are again Topics. Topic Maps do not have a specialized primitive for representing instances. Any instance of a topic type can act as a topic type itself.

The bioinformatics community designed XOL for the exchange of ontologies in the field of bioinformatics. It evolved to become a general language for interchange of ontology and instance data. Being an interchange language, XOL includes primitives found in many knowledge-representation systems, object databases, and relational databases. It provides means to define classes, a class hierarchy, slots, facets, and instances.

RDF (resource description framework) provides a graph-based data model, consisting of nodes and edges. Nodes correspond to objects or *resources* and the edges to properties of these objects. The labels on the nodes and edges are Uniform Resource Identifiers (URIs). However, RDF itself does not define any primitives for creating ontologies. It is the basis for several other ontology-definition languages such as RDFS and DAML+OIL.

RDF Schema[2] defines the primitives for creating ontologies. Figure 1 shows an example of a graph representing our ontology of wines as an RDFS. In RDFS, there are classes of concepts, which constitute a hierarchy with multiple inheritance. For example, the class **Wine** is a subclass of the class **Drink**. Classes typically have instances (for example, a specific red wine is an instance of the **Red Wine** class) and a resource can be an instance of more than one class (for example, **Romariz Port** is an instance of both the **Red Wine** and the **Dessert Wine** classes). Resources have properties associated with them (for example, **Wine** has **flavor**). Properties describe attributes of a resource or a relation of a resource to another resource. RDFS defines a property's *domain*—resources that can be subjects of the property—and a property's *range*—resources that can be objects of the

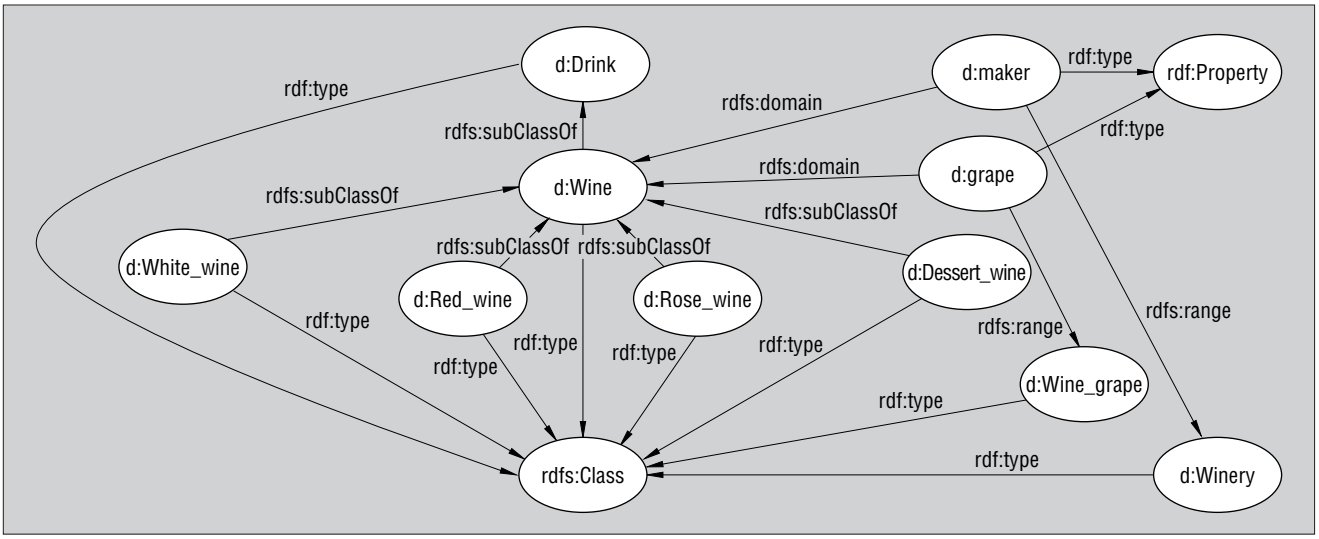


Figure 1. An RDF Schema graph representing the Wine ontology.

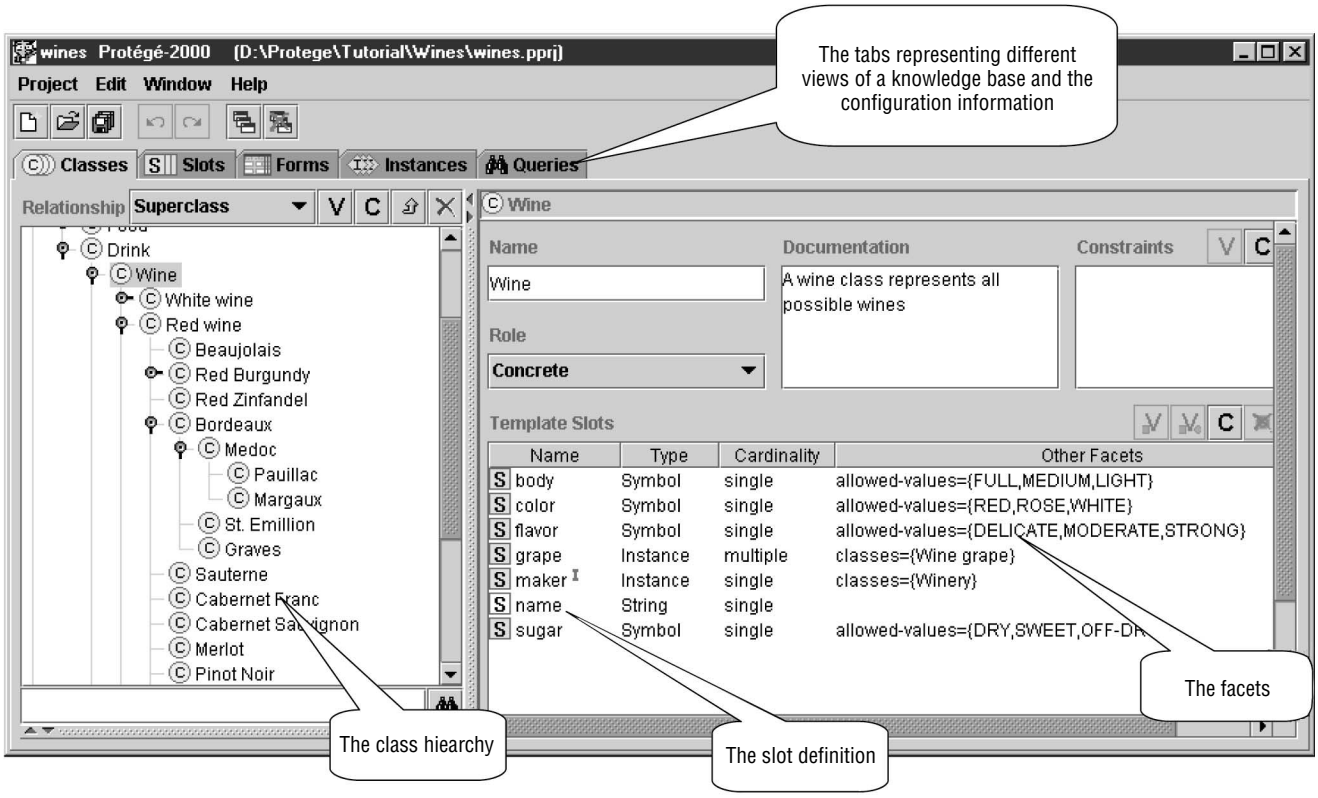


Figure 2. A snapshot of the ontology representing wines. The tree on the left represents a class hierarchy. The form on the right shows the definition of the Wine class.

property. For example, the property **maker** may have a class **Wine** as its domain and a class **Winery** as its range.

DAML+OIL (DARPA Agent Markup language + Ontology Inference Layer)[3] takes a different approach to defining classes and instances. In addition to defining classes and instances declaratively, DAML+OIL and other description-logics languages let us create *intensional* class definitions using Boolean expressions and specify necessary, or necessary and sufficient, conditions for class membership. These languages rely on inference engines (classifiers) to compute a class hierarchy and to determine class membership of instances based on the properties of classes and instances. For example, we can define a class of Bordeaux wines as "a class of wines produced by a winery in the Bordeaux region." In DAML+OIL, we can also specify global properties of classes and slots. For example, we can say that the **location** slot is *transitive*: if a winery is located in the Bordeaux region and the Bordeaux region is located in France, then the winery is in France. We will describe DAML+OIL in more detail later.

We can see from this discussion that Semantic Web languages for representing ontologies and instance data have many features in common. At the same time, there are significant differences stemming from different design goals for the languages. In adapting Protégé-2000 as an editor for these languages, we build on the similarities among them and custom-tailor the tool to account for the individual differences.

Protégé-2000

For many years now, experts in domains such as medicine and manufacturing have used Protégé-2000 for domain modeling. We show not only how we adapt Protégé-2000 to the new world of the Semantic Web—reusing its user interface, internal representation, and framework—but also how our changes enable conceptual modeling with the new Semantic Web languages.

Protégé-2000 is highly customizable, which makes its adaptation as an editor for a new language faster than creating a new editor from scratch. The following features make this customization possible:

- *An extensible knowledge model.* We can redefine declaratively the representational primitives the system uses.
- *A customizable output file format.* We can implement Protégé-2000 components that translate from the Protégé-2000 internal representation to a text representation in any formal language.
- *A customizable user interface.* We can replace Protégé-2000 user-interface components for displaying and acquiring data with new components that fit the new language better.
- *An extensible architecture that enables integration with other applications.* We can connect Protégé-2000 directly to external semantic modules, such as specific reasoning engines operating on the models in the new language.

Protégé-2000 knowledge model

Protégé-2000's representational primitives—the elements of its *knowledge model*[4]—are very similar to those of the Semantic Web languages that we described earlier. Protégé-2000 has classes, instances

of these classes, slots representing attributes of classes and instances, and facets expressing additional information about slots.

Figure 2 shows an example definition of a class, which is part of an ontology describing wines, food, and desirable wine–food combinations. In the figure, the tree on the left represents a class hierarchy. The class of Pauillac wines, for instance, is a subclass of the class of Médoc wines. In other words, each Pauillac wine is a Médoc wine. The class of Médoc wines is, in turn, a subclass of Red Bordeaux wines and so on. The form on the right in Figure 2 represents the definition of the selected class (Wine). There is the name of the class, its documentation, a list of possible constraints, and definitions of slots that the instances of this class will have. Instances of the class Wine will have slots describing their flavor, body, sugar level, the winery that produced the wine, and so on.

The form in Figure 3 displays an instance of the class Pauillac representing Château Lafite Rothschild Pauillac, and the fields display the slot values for that instance. Therefore, we know that Château Lafite Rothschild Pauillac has a full body and strong flavor among other properties. Both the class-definition forms (the right-hand side in Figure 2) and the instance-definition forms (Figure 3) are *knowledge-acquisition forms* in Protégé-2000. The fields on the knowledge-acquisition forms correspond to slot values, and we define classes and instances by filling in slot values in these fields. Protégé-2000 generates knowledge-acquisition forms automatically based on the types of the slots and restrictions on their values.

The Protégé-2000 user interface (Figure 2) consists of several *tabs* for editing different elements of a knowledge base and custom tailoring the layout of the knowledge-acquisition forms, such as the forms in Figures 2 and 3. The Classes tab defines classes and slots, and the Instances tab acquires specific instances. The Forms tab allows us to change the layout and the contents of the knowledge-acquisition forms.

We can customize almost all of the Protégé-2000 features we have described to fit a specific domain or task by

- changing declaratively the standard class and slot definitions;
- changing the content and the layout of the knowledge-acquisition forms; and
- developing plug-ins using the Protégé-2000 application-programming interface.[5]

Let's look at how we can customize Protégé-2000 and then see how we can use this flexibility to create Protégé-based editors for new Semantic Web languages.

Changing the notion of classes and slots

The definition of the Wine class in Figure 2 is a standard class definition in Protégé-2000, with a class name, documentation, list of slots, and so on. What if we need to add more attributes to a class definition, or change how a class looks, or change the default definition of a class in the system? For instance, we might want to add a list of a few best wineries for each type of wine in the hierarchy. Such a list is a property of a class (such as Pauillac wines) rather than a property of specific instances of the class (such as Château Lafite Rothschild Pauillac). The list of the best wineries for a class is not inherited by its subclasses: The best wineries producing red Bordeaux are not necessarily the same as the best Médoc or Pauillac wineries (although, they may overlap). Therefore, this list must become a part of a class definition the same way as documentation is a part of a class definition. The Protégé-2000 *metaclass* architecture lets us do just that.[4,5]

Metaclasses are templates for classes in the same way that classes are templates for instances. We can define our own metaclasses and effectively change a definition of what a class is, in the same way we would define a new class. The default Protégé-2000 template (the standard metaclass) defines the fields that we see in Figure 2. We can extend declaratively this standard definition of what a class is with new fields of any type by defining a new metaclass, which simply becomes a part of the knowledge base. Figure 4 shows a definition of the Red Bordeaux class that includes the additional field with a list of the best Bordeaux wineries.

Similarly, we can define new *metaslots* as user-defined templates for new slots. If slot definitions in our system must have fields in addition to the ones that Protégé-2000 has, we simply define new templates where we describe these new fields.

Custom-tailoring slot widgets for value acquisition

The look and behavior of the fields on the knowledge-acquisition forms in Figures 2 and 3 depend on the types of the values that the fields can take. A field for a string value, such as a class name, has a simple text window. A field that contains a list of complex values is a list box with buttons to add and remove values and to create new values. These fields are called *slot widgets*. They not only display the values appropriately, but also help to ensure that the values are correct based on the slot definitions in the ontology. For example, the maker of a wine must be a winery—an instance of the Winery class. The slot widget for the maker slot in Figure 3 lets us set the value only to a Winery instance.

Developers can extend Protégé-2000 by implementing their own slot widgets that are tailored to acquire and verify particular kinds of values. Suppose we wanted to be more precise about the sugar level in wine and to mark it on a scale rather than simply choosing among three values—dry, sweet, or off-dry.

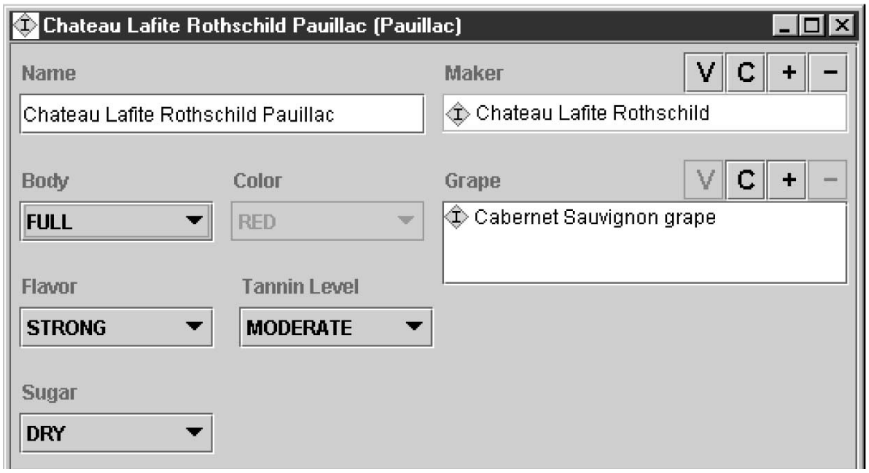


Figure 3. An instance of the class Pauillac representing the Château Lafite Rothschild Pauillac. This wine has a full body, a strong flavor, and a moderate tannin level, among other properties.

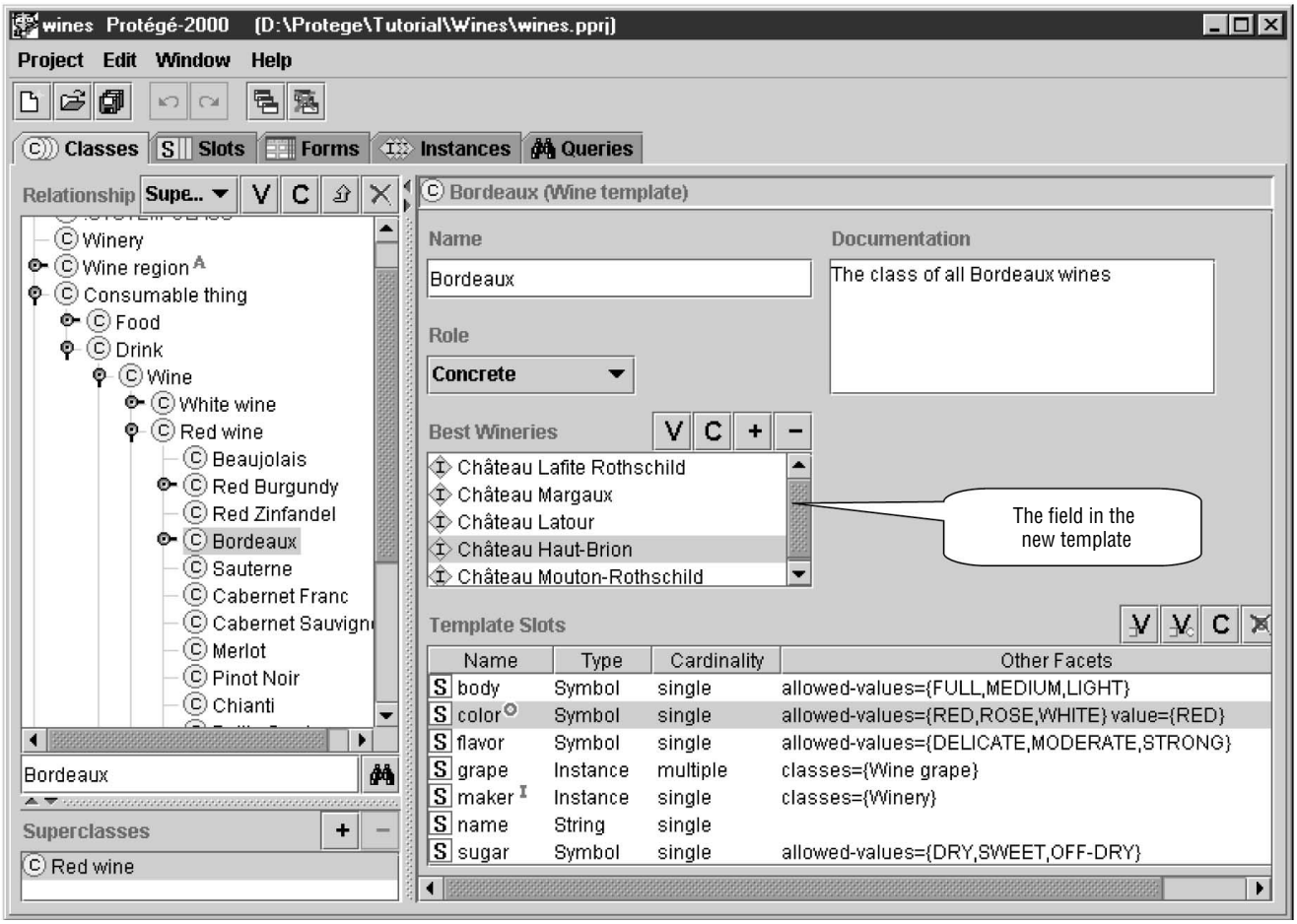


Figure 4. A class definition that uses a nonstandard template. We added the best wineries slot to the standard class-definition template.

We could store the value as a number in the sugar slot. We could use a slot widget that would let us select the value on a slider rather than enter a number (see Figure 5). When we customize knowledge-acquisition forms, we choose not only the layout of the fields on the form, but also the slot widgets that must be used for different fields. The slot widgets we choose do not usually affect the contents of the knowledge base itself, but their use can make the look and feel of the tool much more suitable for a particular domain or language. Slot widgets also can help ensure the internal consistency of a knowledge base by checking, for example, that an integer value that we enter is between the allowed maximum and minimum for that slot.

Using a back-end plug-in to generate the right output

When we develop a domain model in Protégé-2000, we do not need to think about the specific file format that Protégé-2000 will use to save our work. We think about our domain at the conceptual level and create classes, slots, facets, and instances using the graphical user interface. Protégé-2000 then saves the resulting domain models in its own file format. Developers can change this file format in the same way they plug in slot widgets. Back-end plug-ins let developers substitute the Protégé-2000 text file format with any other file format. For example, suppose we wanted to use XML to publish the wine ontology and other domain models we create using Protégé-2000. We would then need to create an XML back end that substitutes files in the Protégé-2000 format with the files in XML. A back end creates a mapping between the in-memory representation of a Protégé-2000 knowledge base and the file output in the required format. The back end also enables us to import the files in that format into Protégé-2000. The new file format has the same status as the Protégé-2000 native file format, and the users can choose either format for their files.

Editing Semantic Web languages with Protégé-2000

Armed with the arsenal of tools to custom-tailor Protégé-2000 quickly and easily, let's look at what is involved in creating a Protégé-2000 editor for a new Semantic Web language. We will use the Protégé-RDFS editor developed in our laboratory as an example, but the ideas are the same for any new language.

We start creating a Protégé-2000 editor for our new language by determining the differences between the knowledge models of the two languages: the knowledge model of Protégé-2000 and the knowledge model underlying our language of choice. We then decide which of the available tools—metaclasses, custom user-interface components, or a custom back end—we will use to reconcile the differences or to hide them from the user.

In practice, the overlap between the knowledge models underlying the Semantic Web languages available today is very large. The models might use different terminology for

the same notion (for example, *slots* in Protégé-2000 and *properties* in RDFS). However, the structure of the concepts, the underlying semantics, and the restrictions are often similar.

When we compare the two knowledge models, we identify four categories of concepts (see Figure 6):

1. *Concepts that are exactly the same in the two languages (possibly with different names)*. Usually, classes, inheritance, instances, slots as properties of classes and instances, and many of the slot restrictions fall into this category.
2. *Concepts that are the same but expressed differently in the two languages*. For example, Protégé-2000 associates slots with classes by attaching a slot to a class. RDFS defines essentially the same relationship by defining a domain of a property.
3. *Concepts in our language of choice that do not have an equivalent in Protégé-2000*. For example, RDFS allows an instance to have more than one type, whereas in Protégé-2000 each instance has a unique direct type.
4. *Concepts that Protégé-2000 supports and our language of choice does not*. For example, Protégé-2000 allows a slot to have more than one allowed class for its values, whereas the range of a property in RDFS is limited to a single class.

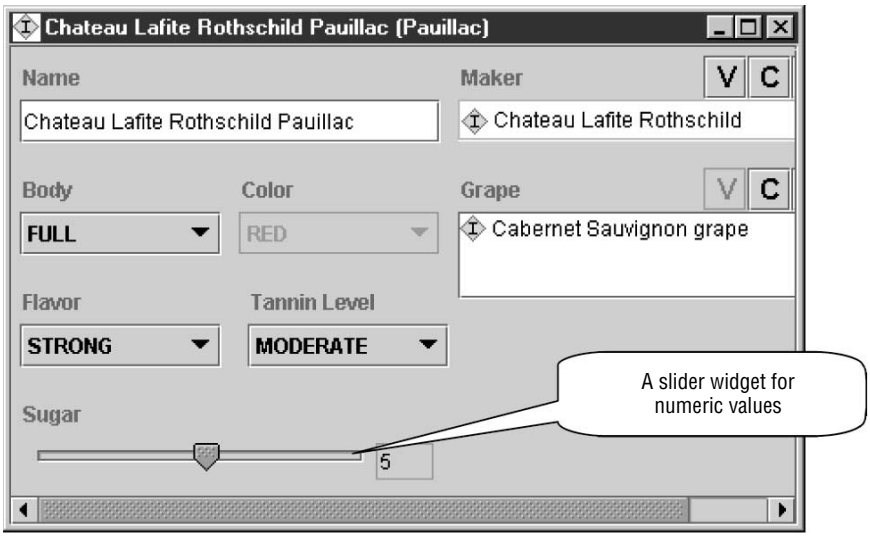
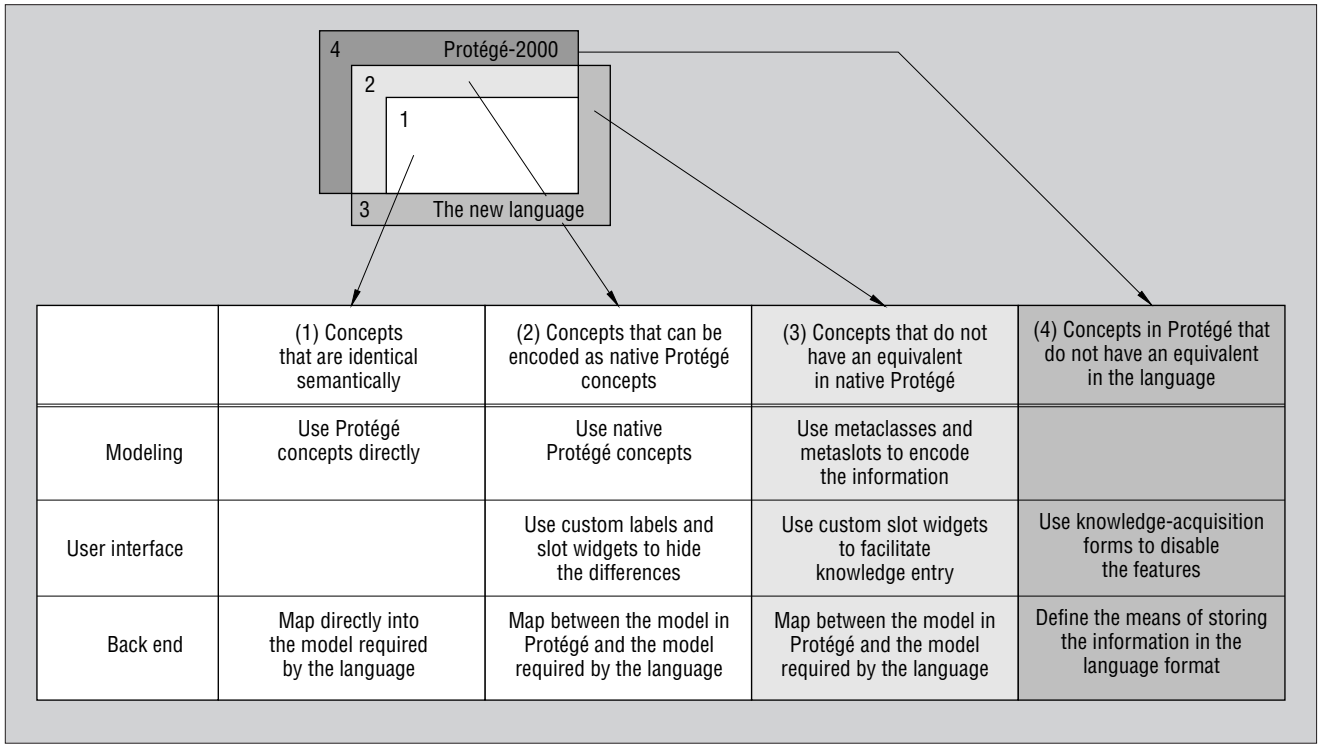


Figure 5. Changing a slot widget. We use a slider instead of a simple field to acquire numeric values for the sugar level.

Naturally, we can express all the features of our language that fall into the first category directly in Protégé-2000. We deal with the differences in the other three categories by defining appropriate metaclasses and metaslots and by resolving the remaining changes in the back end. We hide the differences from the user behind custom-tailored slot widgets.

The second item on the list, the concepts that do not have a direct equivalent in Protégé-2000 but that can be mapped to native Protégé-2000 concepts, deserves a special discussion. Consider domains of properties in RDFS (rdfs:domain). A domain specifies a class on which a property might be used. For example, the domain of the flavor property is the Wine class. Protégé-2000 slots are similar to properties in RDFS. Attaching a slot to a class in Protégé-2000 also specifies that a slot can be used with that class. For example, the flavor slot



	(1) Concepts that are identical semantically	(2) Concepts that can be encoded as native Protégé concepts	(3) Concepts that do not have an equivalent in native Protégé	(4) Concepts in Protégé that do not have an equivalent in the language
Modeling	Use Protégé concepts directly	Use native Protégé concepts	Use metaclasses and metaslots to encode the information	
User interface		Use custom labels and slot widgets to hide the differences	Use custom slot widgets to facilitate knowledge entry	Use knowledge-acquisition forms to disable the features
Back end	Map directly into the model required by the language	Map between the model in Protégé and the model required by the language	Map between the model in Protégé and the model required by the language	Define the means of storing the information in the language format

Figure 6. Comparing the knowledge models of Protégé-2000 and a new Semantic Web language.

Table 2. Creating the Protégé-based RDFS editor.

Category	(1) Concepts in RDFS that are (nearly) identical to Protégé concepts	(2) Concepts in RDFS that can be encoded as native Protégé concepts	(3) Concepts in RDFS that do not have an equivalent in native Protégé	(4) Concepts in Protégé that do not have an equivalent in RDFS
Modeling	rdfs:Class = :STANDARD-CLASS rdfs:subClassOf = subclass of rdf:type = instance of rdf:Property = :STANDARD-SLOT rdfs:subPropertyOf = subslot of rdfs:Resource = :THING rdf:comment = :DOCUMENTATION	Do not use explicit rdfs:domain and rdfs:range for properties; rdfs: domain encoded as slot attachment; rdfs:range encoded as allowed class Instance-typed slots	rdfs:Class is a default for metaclass, and rdf:Property is a default metaslot; add properties rdfs:isDefinedBy, rdfs:seeAlso as core slots; add rdfs:ConstraintProperty and rdfs:ConstraintResource as core classes; multiple types of an instance	Cardinality, inverse slot, and default value facets; multiple allowed classes for a slot
User interface	Custom labels on class and slots forms (for example, "Properties" and "Comment")		Plug-in URI slot widget for validating URI-type slots.	Disable default-value and inverse-slot widgets on slot forms.
Back end	Map Protégé concepts directly to RDFS concepts	Translate slot attachments as rdfs:domain for properties and allowed classes as rdfs:range	On import, create new class as a subclass of the multiple types.	Write out extra facet information as Protégé-specific properties on properties. If a slot has multiple allowed classes, create a new class for rdfs:range value. On import, do the reverse.

is attached to the **Wine** class. We have two ways to encode the RDFS domain information in a Protégé-RDFS editor. First, we can add a **domain** slot to a template (metaslot) that we will use for all our slots. Then, a field for **domain** will appear on each form for a slot, and we will fill it in there. Second, we can simply use the native Protégé-2000 notion of slot attachment and translate the attachments of slots to classes into domains of properties in the back end. The second solution lets us use the Protégé-2000 user interface directly and hides the features of a specific language used to store the information.

We find it extremely beneficial to adopt the paradigm of using the native Protégé-2000 features wherever possible and of resorting to additional definitions, such as metaclasses and metaslots, only when absolutely necessary. This approach maximally facilitates the exchange of domain models among different languages, which we edit (or will edit) with Protégé-2000. As new languages emerge and we experiment with them, the knowledge models underlying these languages will undoubtedly overlap. By encoding as much as possible in the native Protégé-2000 structures and leaving part of the translation between the Protégé-2000 model and the language to the back end, we maximize the amount of information that we will preserve by simply loading a knowledge base in one language supported by Protégé-2000 and saving it to another language. Even though there would often be some parts of these models that will not be part of this overlap, we are maximizing the amount of information that gets ported among models in different languages for free.

Having generated the four groups of concepts after comparing the two knowledge models (see Figure 6), we can reconcile the differences using

1. modeling—by changing default definitions of classes and slots at the modeling level;
2. the user interface—by developing specialized user-interface components; and
3. the back end—by implementing the new back end that will translate between the domain model in Protégé-2000 and the domain model in our language of choice.

Let's look at how each of these three levels works using the development of a Protégé-based RDFS editor as an example (see Table 2 for a summary of the entire process).

The modeling level

We start by determining which concepts in our language of choice are identical to Protégé-2000 concepts or that can be represented using the native Protégé-2000 concepts. We use the native Protégé-2000 as a means to model this group of concepts, even if it is not how these elements are directly expressed in our language. We then define the new templates for class and slot definitions if necessary.

Consider, for example, the two attributes—**rdfs:seeAlso** and **rdfs:isDefinedBy**—that are associated with each class and each property in RDFS. The **rdfs:seeAlso** property specifies another resource containing information about the subject resource, and the **rdfs:isDefinedBy** property indicates the resource defining the subject resource. The values of these properties are other resources or URIs pointing to other resources. We must add these two fields that the Protégé-2000 itself does not have to each class and slot form in our knowledge base. To add these fields, we define a new metaclass that will serve as a template for RDFS classes. This metaclass is, in fact, equivalent to the RDFS class **rdfs:Class**. Figure 7 shows the definition of **rdfs:Class** with the new template slots that will appear on each class form that uses this template.

The user-interface level

When creating a Protégé-based editor for a new language, we can change both the behavior and the look and feel of the knowledge-acquisition forms to reflect the terminology and the features of the language. First, we can change the labels on the forms—the simplest type of customization. For example, we can easily replace Protégé's "Template slots" label in a class definition with the RDFS "Properties" label to give the form an RDFS look. Other elements that we can easily configure by manipulating the forms include whether or not a field should be visible to the

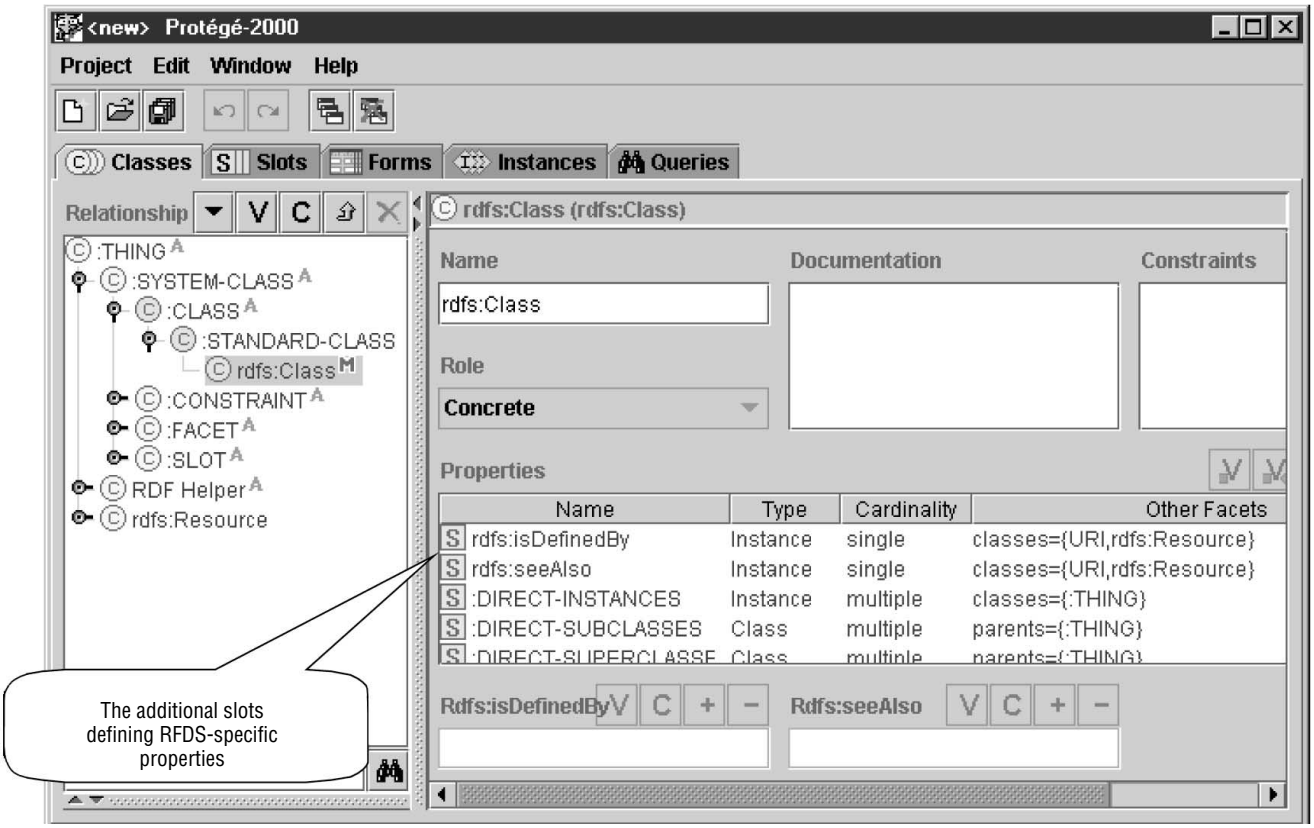


Figure 7. A template definition for classes in RDFS. The class rdfs:Class inherits most of the slots from the standard class template, but the two slots at the top of the list of properties are the ones that we defined for RDFS.

user, the buttons on the fields, the position and size of the fields, and the slot widgets to be used for each field. We perform this configuration entirely in the Protégé-2000 Forms tab and not in the programming code.

We could also develop our own slot-widget plug-ins to allow editing and verification of elements that are unique to our language. For example, a URI widget in the Protégé-RDF editor can validate that the user has entered a correct URI or even take the user to the corresponding Web page.

Disabling fields for some slots on the form prevents the user from exercising Protégé-2000 features that the particular Semantic Web language does not support. For example, we can disable the field for entering default slot values in the Protégé-RDFS editor, because RDFS does not support default values.

The back-end level

Whatever the differences between Protégé-2000 and our language that we could not resolve at the modeling and user-interface levels, we will need to reconcile in the module that saves the internal Protégé-2000 representation in the required output file format—the back-end plug-in. The back-end plug-in

1. saves a Protégé-2000 knowledge base in a file format that conforms to the syntax of our language of choice;
2. maps the elements of the Protégé-2000 knowledge base that do not have a direct equivalent in our language to the appropriate set of elements in this language; and
3. imports domain models in this language that were developed elsewhere for editing in Protégé-2000.

Usually, when developers define a language with a new syntax, they quickly implement a parser that allows developers to read and write files in that language's syntax. Many of the new languages are extensions of XML or RDF, and thus we can often use the existing XML and RDF parsers to take care of the syntactic part of adapting to the new language.

In RDFS, the back end must deal with a number of issues that we did not resolve at the modeling level or in the user interface. We might have resolved some of these issues there, but it would have unnecessarily complicated the editor for the user. For example, instances in Protégé-2000 are of a single class, whereas in RDFS they can be direct instances of several classes (for example, they have several direct types). Because the RDFS model is more general, we have no problem in saving a Protégé-2000 knowledge base in RDFS. However, when we import RDFS instance data into Protégé-2000, we must deal with instances that have several direct types. Suppose we have a class for red wines and a class for dessert wines. We have Romariz Port as an instance of both classes in RDFS. When we import this RDFS instance data into Protégé-2000, the back end can create a new class that is a subclass of both classes (for example, denoting a concept of dessert red wines) and make the Romariz Port instance an instance of this new class. We can record the two original classes of Romariz Port as additional slots on the newly created class (as shown in Figure 4). When saving back to RDFS, the back end can extract the information from this slot, thus preserving the original model.

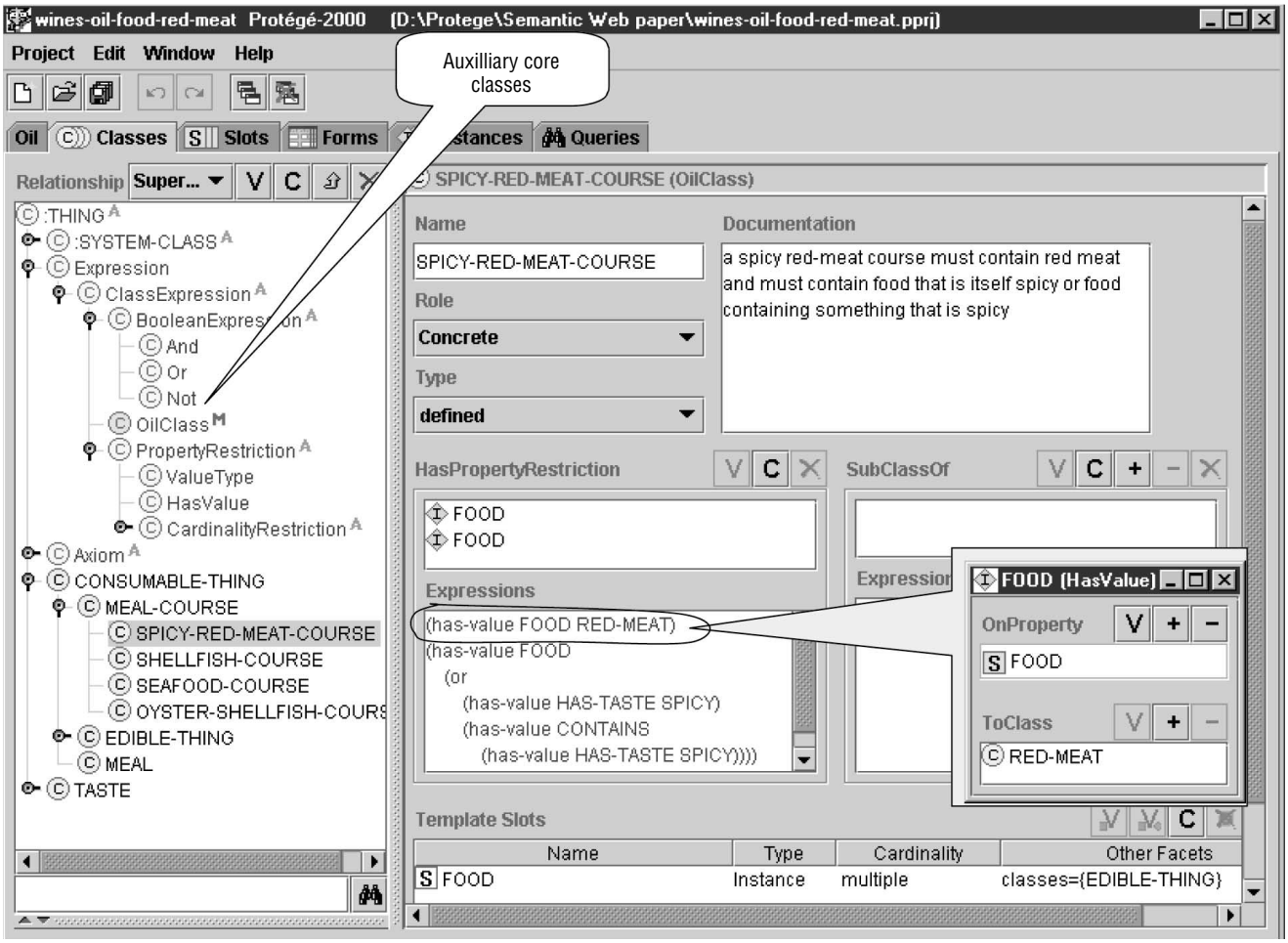


Figure 8. The definition for the spicy-red-meat-course class in the Protégé-OIL editor. In addition to the standard fields, such as those shown in Figure 2), we have OIL-specific fields such as hasPropertyRestriction and subClassOf for specifying complex class expressions. These slots use the OIL-specific slot widget to display expressions. The tree on the left contains the auxiliary core classes we defined for OIL.

Any user-defined back end has the same status as all the other back ends, including the ones that are part of the core Protégé-2000 system: it can be used as a storage format for Protégé-2000 knowledge bases. Therefore, there is another, no less important, goal of a back-end plug-in: to ensure that when we create a knowledge base in Protégé-2000, save it using the back-end plug-in, and load it again, we have preserved all the information. Hence, we must find a way to store the elements that Protégé-2000 supports, but that our language of choice does not. Most Semantic Web languages are flexible enough to easily store this information. For example, in RDFS, we simply add new Protégé-specific properties to slots to record default values, which RDFS does not have. These properties have no meaning to another RDFS agent, but if we read the knowledge base back in Protégé-2000, we will have the default values preserved.

Creating new tabs to include other semantic modules

In addition to creating a Protégé-based editor for a new Semantic Web language, developers can plug in other applications in the knowledge-base–editing environment. In addition to the standard *tabs* that constitute the Protégé-2000 user interface (Figures 2 and 4), developers can create *tab plug-ins* in the same way they can plug in new slot widgets. These tabs can include arbitrary applications that benefit from the live connection to the knowledge base. These applications then become an integral part of the knowledge-base–editing environment.

Consider our wine example again. Having created a knowledge base of wines and food and the appropriate combinations, we might want to build an application that produces wine suggestions for a meal course in a restaurant. Such an application would actively use the data in the knowledge base but it would also implement its own reasoning mechanism to analyze suggestions. We can implement this wine-selection application as a tab plug-in.

In practical terms, a tab plug-in is a separate application, a developer's own user-interface space from which the developer can connect to, query, and update the current Protégé-2000 knowledge base.

In the realm of the Semantic Web, a tab can include any applications that would help us acquire or annotate the knowledge base. Such applications can

- enable direct annotation of HTML pages with semantic elements;

- provide connection to external reasoning and inference resources;
- acquire the semantic data automatically from text; and
- present a graphical view of a set of interrelated resources.

Using Protégé to edit DAML+OIL

DAML+OIL, the Semantic Web language that was heavily inspired by research in description logics (DL), allows more types of concept definitions in ontologies than Protégé-2000 and RDFS do. The DL-inspired languages usually include the following features in addition to the ones found in the traditional frame-based languages:

- We can specify not only *necessary* but also *sufficient* conditions for class membership. For example, if a wine is produced by a winery from the Bordeaux region, it is a Bordeaux wine.
- We can use arbitrary Boolean expressions in class and slot definitions to specify superclasses of a class, domain and range of a slot, and so on. For example, a spicy red-meat course must contain red meat and must contain food that is itself spicy *or* food containing something that is spicy.
- We can specify global slot properties. For example, location is a *transitive* property: if the Château Lafite Rothschild winery is in the Bordeaux region and the Bordeaux region is in France, then the Château Lafite Rothschild winery is in France.
- We can define global axioms that express additional properties of classes. For example, we can say that the classification of the class of all wines into the subclasses for red, white, and rosé wines is *disjoint*: Each instance of the Wine class belongs only to one of these subclasses.

We have adapted Protégé-2000 to work as an OIL editor. (The OIL language is a precursor for DAML+OIL.) In doing so, we followed the same steps we described in creating the Protégé-based RDFS editor. In addition, we have integrated external services for OIL ontologies into Protégé-2000. Integrating DAML+OIL would require nearly identical steps.

The modeling level

We introduce the new class and slot templates, OilClass and OilProperty, to specify complex class and slot definitions. As a result, a class template, for example, acquires these three new fields (see Figure 8):

1. type—to specify whether the class definition contains only necessary or both necessary and sufficient conditions for class membership;
2. hasPropertyRestriction—to specify complex expressions for slot restrictions; and
3. subclassOf—to specify complex expressions describing the position of the class in the class hierarchy.

To integrate OIL into Protégé-2000, we used the names from the RDFS serialization syntax of (Standard-)OIL and not the plain ASCII version. See www.ontoknowledge.org/oil/ for the various syntaxes and versions.

Just as for RDFS, we use as many native Protégé-2000 mechanisms for modeling OIL ontologies as possible. If a new class is simply a subclass of several existing classes in the hierarchy, we use Protégé's own notion of subclasses by placing the new class where it belongs in the hierarchy. However, if a superclass definition requires boolean expressions—something Protégé-2000 does not allow—we use the subClassOf field that we see on the template. Even though Protégé-2000 does not understand the semantics of this field, we can represent this additional superclass information declaratively, and then pass it to a classifier or simply save in OIL.

We use the hasPropertyRestriction field when we need complex expressions or when we need to specify *existential* slot constraints: Protégé-2000 allows definition of value-type constraints on slots ("All values of this slot must be instances of this class"). OIL allows existential slot constraints in addition to the value-type constraints ("One value of this slot must come from this class and one value must come from that class"). We build the complex expressions declaratively by creating instances of core auxiliary classes. We can see some of these core classes in the tree in Figure 8. In the example, we specify a subclass of a meal-course, spicy-red-meat-course, which we define as "a course that must contain red meat *and* must contain food that is itself spicy *or* food containing something that is spicy."

Even though Protégé-2000 does not support some of the semantics that OIL has, we can still encode the additional information declaratively. Protégé-2000 will "ignore" the information, but it will be able to pass it on to a classifier or to encode it in OIL so that an OIL agent can understand it.

The user-interface level

Apart from changing labels and rearranging fields on the forms for the OilClass and OilProperty templates, we created a new slot widget to allow easier editing of nested expressions such as the ones representing "food that is itself spicy *or* food containing something that is spicy" in Figure 8. This widget augments the standard Protégé-2000 widget for selecting and creating values for instance-valued slots with a display of the nested expressions in a more practical form. A further extension of this simple but effective slot widget can include a full context-sensitive, validating expression editor.

The back-end level

We describe here an OIL back end that produces the RDFS output for OIL. Therefore, we can build largely on the existing RDFS back end. In defining the class and slot names and the structure of the auxiliary core classes in the OIL editor, we have mainly adhered to the RDFS specification of OIL. As a result, just using the RDFS back end, described earlier, gives us an output that is very close but not identical to the RDFS OIL output that we need. Thus, to create the OIL back end, we started with the existing RDFS back end. We adapted it to add the parts of definitions specified by the native Protégé-2000 means to the complex class expressions.

The OIL back end encodes the concepts that Protégé-2000 has and OIL does not (global cardinality restrictions on slots, for example) by defining additional statements in a Protégé namespace. An OIL agent will not understand these statements and will ignore them, but Protégé-2000 will be able to extract the necessary information from them.

Because many Semantic Web languages are in their infancy and already come in many different versions, there is an alternative approach to developing specific back ends for each of these versions. We can create a general RDF back end for Protégé-2000 and then use a declarative and easily adaptable RDF transformation language for generating the desired outputs. Some research groups are currently investigating such a back end and the corresponding RDF transformation (and query) language.

Accessing external services through a tab plug-in

The DL languages, such as OIL and DAML+OIL, traditionally rely on an inference component—a classifier—to find the

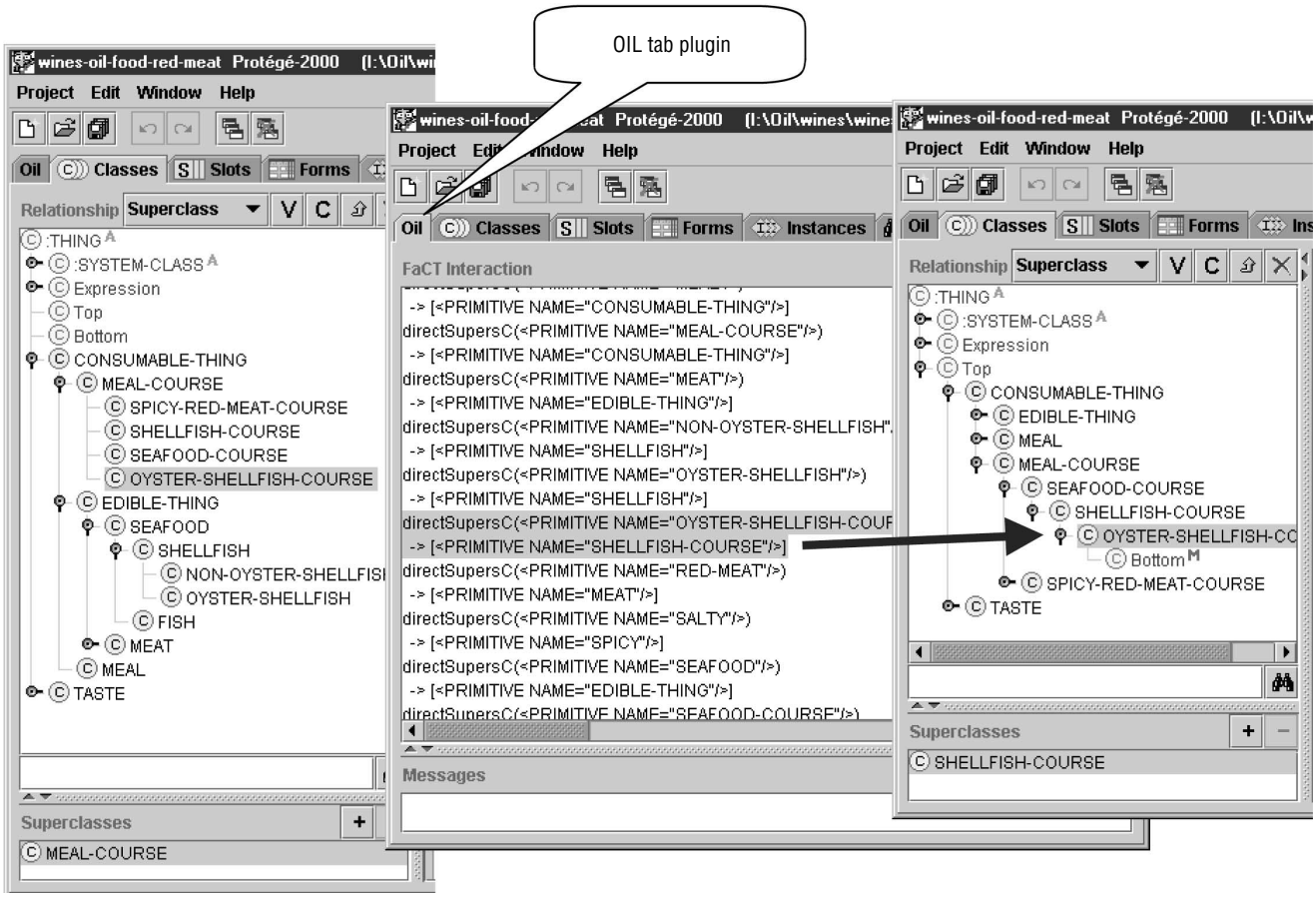


Figure 9. Tab plug-in for classification of OIL ontologies. On the right, we see a hierarchy of meal courses before classification. The middle pane shows interactions with the FaCT classifier. The hierarchy on the right is the one that the classifier computed.

right position of a class in the class hierarchy and to determine which class definitions are unsatisfiable (cannot have any instances). Therefore, it is crucial to have a connection to a DL classifier as part of the environment for editing OIL and DAML+OIL ontologies. Having created a set of definitions, we can invoke the classifier to determine how the evolving class hierarchy looks. We can see the effects that changes in class definitions will have on the evolving hierarchy. We can immediately check if logical expressions defining a class contradict one another making the class unsatisfiable.

Therefore, in order to create a full-fledged Protégé-based OIL editor, we need to connect Protégé-2000 to such an inference component and present the results to the user. We implemented this connection as a Protégé-2000 tab plug-in.

Figure 9 shows the OIL tab in action. Initially, the class hierarchy has the various **meal-course** subclasses as siblings. In addition, we specify that an **oyster-shellfish-course** is a **meal-course** that has OYSTERS as the value for its FOOD slot; a **shellfish-course** is a **meal-course** that has **shellfish** as its **food**, and so on. We then use the OIL tab to connect to a DL classifier, FaCT,[6] and to have it rearrange the class hierarchy according to the class definitions. In the resulting hierarchy, the **oyster-shellfish-course** class, for example, is correctly classified as being a subclass of the **shellfish-course** class.

With the advent of the Semantic Web, the current network of online resources is expanding from a set of static documents designed mainly for human consumption to a new Web of dynamic documents, services, and devices, which software agents will be able to understand. Developers will likely create many different representation languages to embrace the heterogeneous nature of these resources. Some languages will be used to describe specific domains of knowledge; others will model capabilities of services and devices. These languages will have different emphasis, scope, and expressive power.

Protégé-2000 provides full-fledged support for knowledge modeling and acquisition. Developers also can custom-tailor Protégé-2000 quickly and easily to be an editor for a new Semantic Web language. A Protégé-based editor enables modeling at a conceptual level that allows developers to think in terms of concepts and relations in the domain that they are modeling and not in terms of the syntax of the final output.

By adapting Protégé-2000 to edit a new Semantic Web language rather than creating a new editor from scratch or using a text editor to create ontologies in the new language, we obtain a graphical, conceptual-level ontology editor and knowledge-acquisition tool. We get a new editor to experiment with the new language without investing many resources into it. And we can use Protégé-2000 as an *interchange module* to translate most of the models in other Semantic-Web languages into our new language and vice versa. In our experience, it takes a few days

to adapt Protégé-2000 to a new Semantic-Web language—a lot less time than is required to create any sophisticated software from scratch. We were able to create these editors even for a language like OIL, which takes a knowledge-modeling approach that is different from the frame-based approach for which Protégé originally was designed. The extensible and flexible knowledge model and the open plug-in architecture of Protégé-2000 constitute the basis for developing a suite of conceptual-level editors for Semantic Web languages. ■

Acknowledgments

For more information about the Protégé project, please visit http://protege.stanford.edu. A grant from Spawar, a grant from FastTrack Systems, and the DARPA DAML program supported this work.

References

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

2. D. Brickley and R.V. Guha, "Resource Description Framework (RDF) Schema Specification," World Wide Web Consortium, Proposed Recommendation 1999, www.w3.org/TR/2000/CR-rdf-schema-20000327 (current 28 Mar. 2001).

3. J. Hendler and D.L. McGuinness, "The DARPA Agent Markup Language, " *IEEE Intelligent Systems*, vol. 16, no. 6, Jan./Feb., 2000, pp. 67–73.

4. N.F. Noy, R.W. Fergerson, and M.A. Musen, "The Knowledge Model of Protégé-2000: Combining Interoperability and Flexibility," *Proc. Knowledge Engineering and Knowledge Management: 12th Int'l Conf. (EKAW-2000), Lecture Notes in Artificial Intelligence*, no. 1937, Springer-Verlag, Berlin, 2000, pp.17–32.

5. M.A. Musen et al., "Component-Based Support for Building Knowledge-Acquisition Systems, " *Proc. Conf. Intelligent Information Processing (IIP 2000) Int'l Federation for Information Processing World Computer Congress (WCC 2000)*, Beijing, China, 2000, http://smi-web.stanford.edu/pubs/SMI_Abstracts/SMI-2000-0838.html (current 28 Mar. 2001).

6. I. Horrocks, "The FaCT system," *Proc. Automated Reasoning with Analytic Tableaux and Related Methods: Int'l Conf. Tableaux 98, Lecture Notes in Artificial Intelligence*, no. 1397, Springer-Verlag, Berlin, 1998, pp. 307–312.

The Authors

Natalya F. Noy is a research scientist in the Stanford Medical Informatics laboratory at Stanford University. Her interests include ontology development and evaluation, semantic integration of ontologies, and making ontology-development accessible to experts in noncomputer-science domains. She has a BS in applied mathematics from Moscow State University, Russia, an MA in computer science from Boston University, and a PhD in computer science from Northeastern University in Boston. Contact her at Stanford Medical Informatics, 251 Campus Dr., Stanford Univ., Stanford, CA 94305; noy@smi.stanford.edu.

Michael Sintek is a research scientist at the German Research Center for Artificial Intelligence. Currently, he is project leader of the FRODO project where an RDF-based framework for building distributed organizational memories is developed. He has a Diplom in computer science and economics from the University of Kaiserslautern. Contact him at the German Research Center for Artificial Intelligence (DFKI) GmbH, Knowledge Management Group, Postfach 2080, D-67608 Kaiserslautern, Germany; sintek@dfki.uni-kl.de.

Stefan Decker is a postdoctoral fellow at the Department of Computer Science at Stanford University, where he works on Semantic Web Infrastructure in the DARPA DAML program. His research interests include knowledge representation and database systems for the Web, information integration, and ontology articulation and merging. He has a PhD in computer science from the University of Karlsruhe, Germany, where he worked on ontology-based access to information. Contact him at Stanford University, Gates Hall 4A, Room 425, Stanford, CA 94305; stefan@db.stanford.edu.

Monica Crubézy is a postdoctoral researcher in the Stanford Medical Informatics laboratory at Stanford University. Her research focuses on the modeling and integration of libraries of problem-solving methods in the Protégé knowledge-based-system development framework. She graduated from the École Polytechnique Féminine, France, in general engineering and computer science. She has a PhD in computer science from the Institut National de Recherche en Informatique et Automatique in Sophia Antipolis, France. Contact her at Stanford Medical Informatics, 251 Campus Drive,Stanford University, Stanford, CA 94305; crubezy@smi.stanford.edu.

Ray Fergerson is a programmer in the Stanford Medical Informatics laboratory at Stanford University. He has a BS in physics from the Colorado School of Mines and a PhD in experimental nuclear physics from the University of Texas in Austin. Contact him at Stanford Medical Informatics, 251 Campus Drive, Stanford University, Stanford, CA 94305; fergerson@smi.stanford.edu.

Mark A. Musen's biography appears in the Guest Editors' Introduction on page 25.

Ontology Learning for the Semantic Web

Alexander Maedche and Steffen Staab, *University of Karlsruhe*

The authors present an ontology-learning framework that extends typical ontology engineering environments by using semiautomatic ontology-construction tools. The framework encompasses ontology import, extraction, pruning, refinement, and evaluation.

The Semantic Web relies heavily on formal ontologies to structure data for comprehensive and transportable machine understanding. Thus, the proliferation of ontologies factors largely in the Semantic Web's success. *Ontology learning* greatly helps ontology engineers construct ontologies. The vision of ontology learning that we propose includes a number of complementary disciplines that feed on different types of unstructured, semistructured, and fully structured data to support semiautomatic, cooperative ontology engineering. Our ontology-learning framework proceeds through ontology import, extraction, pruning, refinement, and evaluation, giving the ontology engineer coordinated tools for ontology modeling. Besides the general framework and architecture, this article discusses techniques in the ontology-learning cycle that we implemented in our ontology-learning environment, such as ontology learning from free text, dictionaries, and legacy ontologies. We also refer to other techniques for future implementation, such as reverse engineering of ontologies from database schemata or learning from XML documents.

Ontologies for the Semantic Web

The conceptual structures that define an underlying ontology provide the key to machine-processable data on the Semantic Web. *Ontologies* serve as metadata schemas, providing a controlled vocabulary of concepts, each with explicitly defined and machine-processable semantics. By defining shared and common domain theories, ontologies help people and machines to communicate concisely—supporting semantics exchange, not just syntax. Hence, the Semantic Web's success and proliferation depends on quickly and cheaply constructing domain-specific ontologies.

Although ontology-engineering tools have matured over the last decade,[1] manual ontology acquisition remains a tedious, cumbersome task that can easily result in a knowledge acquisition bottleneck. When developing our ontology-engineering workbench, OntoEdit, we particularly faced this question as we were asked questions that dealt with time ("Can you develop an ontology quickly?"), difficulty, ("Is it difficult to build an ontology?"), and confidence ("How do you know that you've got the ontology right?").

These problems resemble those that knowledge engineers have dealt with over the last two decades as they worked on knowledge acquisition methodologies or workbenches for defining knowledge bases. The integration of knowledge acquisition with machine-learning techniques proved extremely beneficial for knowledge acquisition.[2] The drawback to such approaches,[3] however, was their rather strong focus on structured knowledge or databases, from which they induced their rules.

Conversely, in the Web environment we encounter when building Web ontologies, structured knowledge bases or databases are the exception rather than the norm. Hence, intelligent support tools for an ontology engineer take on a different meaning than the integration architectures for more conventional knowledge acquisition.[4]

In ontology learning, we aim to integrate numerous disciplines to facilitate ontology construction, particularly machine learning. Because fully automatic machine knowledge acquisition remains in the distant future, we consider ontology learning as semiautomatic with human intervention, adopting the paradigm of balanced cooperative modeling for constructing ontologies for the Semantic Web.[5] With this objective in mind, we built an architecture that combines knowledge acquisition with machine learning, drawing on

1094-7167/01/$10.00 © 2001 IEEE

resources that we find on the syntactic Web—free text, semistructured text, schema definitions (such as document type definitions [DTDs]), and so on. Thereby, our framework's modules serve different steps in the engineering cycle (see Figure 1):

- Merging existing structures or defining mapping rules between these structures allows *importing* and *reusing* existing ontologies. **(**For instance, Cyc's ontological structures have been used to construct a domain-specific ontology.[6])
- Ontology *extraction* models major parts of the target ontology, with learning support fed from Web documents.
- The target ontology's rough outline, which results from import, reuse, and extraction, is *pruned* to better fit the ontology to its primary purpose.
- Ontology *refinement* profits from the pruned ontology but completes the ontology at a fine granularity (in contrast to extraction).
- The target application serves as a measure for validating the resulting ontology.[7]

Finally, the ontology engineer can begin this cycle again—for example, to include new domains in the constructed ontology or to maintain and update its scope.

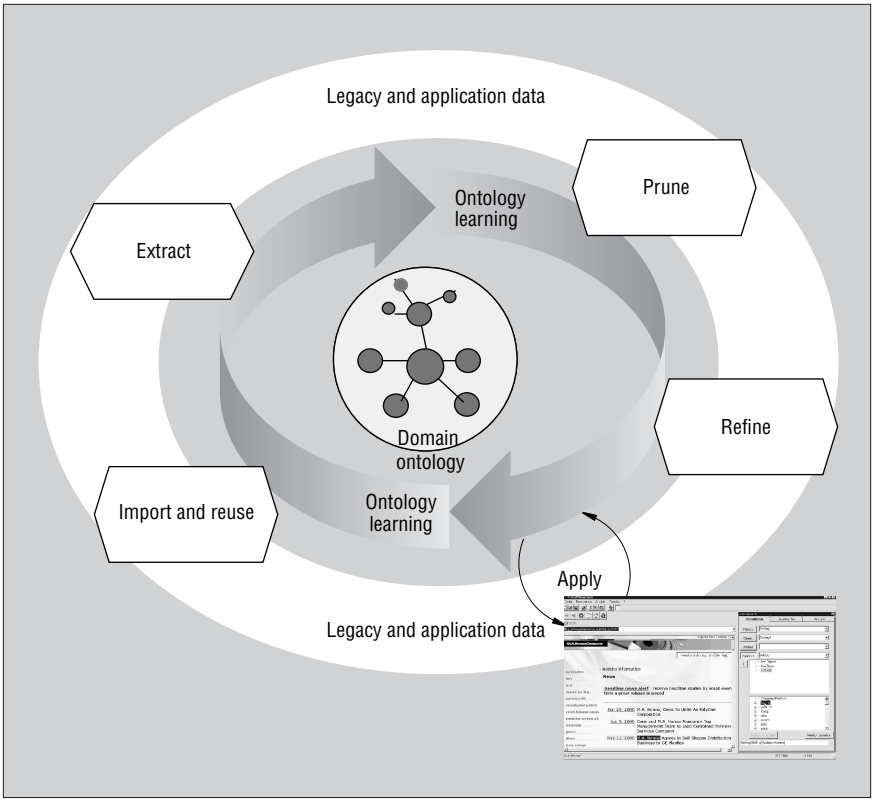


Figure 1. The ontology-learning process.

Architecture

Given the task of constructing and maintaining an ontology for a Semantic Web application such as an ontology-based knowledge portal,[8] we produced support for the ontology engineer embedded in a comprehensive architecture (see Figure 2). The ontology engineer only interacts via the graphical interfaces, which comprise two of the four components: the OntoEdit Ontology Engineering Workbench and the Management Component. Resource Processing and the Algorithm Library are the architecture's remaining components.

The OntoEdit Ontology Engineering Workbench offers sophisticated graphical means for manual modeling and refining of the final ontology. The interface gives the user different views, targeting the epistemological level rather than a particular representation language. However, the user can export the ontological structures to standard Semantic Web representation languages such as OIL (ontology interchange language) and DAML-ONT (*DAML ont*ology language), as well as our own F-Logic-based extensions of RDF(S)—we use RDF(S) to refer to the combined technologies

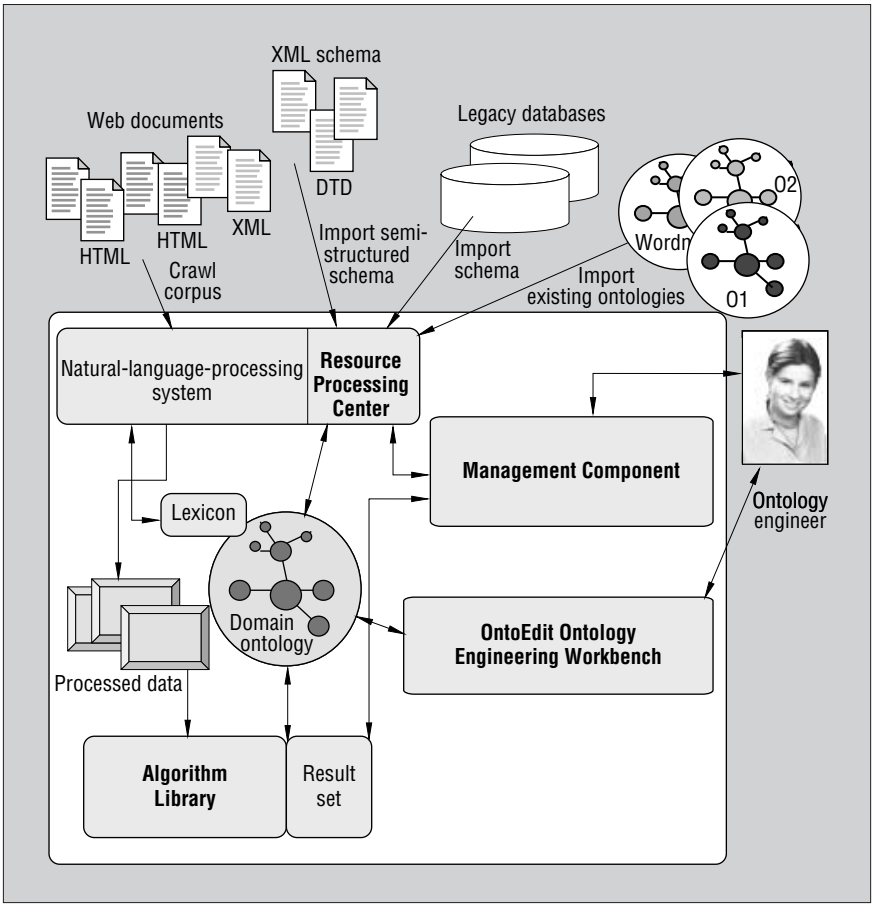


Figure 2. Ontology-learning architecture for the Semantic Web.

of the resource description framework and RDF Schema. Additionally, users can generate and access executable representations for constraint checking and application debugging through SilRi (*si*mple *l*ogic-based *R*DF *i*nterpreter, www.ontoprise.de), our F-Logic inference engine, which connects directly to OntoEdit.

We knew that sophisticated ontology-engineering tools—for example, the Protégé modeling environment for knowledge-based systems[1]—would offer capabilities roughly comparable to OntoEdit. However, in trying to construct a knowledge portal, we found that a large conceptual gap existed between the ontology-engineering tool and the input (often legacy data), such as Web documents, Web document schemata, databases on the Web, and Web ontologies, which ultimately determine the target ontology. Into this void we have positioned new components of our ontology-learning architecture (see Figure 2). The new components support the ontology engineer in importing existing ontology primitives, extracting new ones, pruning given ones, or refining with additional ontology primitives. In our case, the ontology primitives comprise

- a set of strings that describe lexical entries L for concepts and relations;
- a set of concepts C (roughly akin to synsets in WordNet[9]);
- a taxonomy of concepts with multiple inheritance (heterarchy) H_C;
- a set of nontaxonomic relations R described by their domain and range restrictions;
- a heterarchy of relations—H_R;
- relations F and G that relate concepts and relations with their lexical entries; and
- a set of axioms A that describe additional constraints on the ontology and make implicit facts explicit.[8]

This structure corresponds closely to RDF(S), except for the explicit consideration of lexical entries. Separating concept reference from concept denotation permits very domain-specific ontologies without incurring an instantaneous conflict when `merging` ontologies—a standard Semantic Web request. For instance, the lexical entry *school* in one ontology might refer to a building in ontology A, an organization in ontology B, or both in ontology C. Also, in ontology A, we can refer to the concept referred to in English by *school* and *school building* by the German *Schule* and *Schulgebäude*.

Ontology learning relies on an ontology structured along these lines and on input data as described earlier to propose new knowledge about reasonably interesting concepts, relations, and lexical entries or about links between these entities—proposing some for addition, deletion, or merging. The graphical result set presents the ontology-learning process's results to the ontology engineer (we'll discuss this further in the "Association rules" section). The ontology engineer can then browse the results and decide to follow, delete, or modify the proposals, as the task requires.

Components

By integrating the previously discussed considerations into a coherent generic architecture for extracting and maintaining ontologies from Web data, we have identified several core components (including the graphical user interface discussed earlier).

In trying to construct a knowledge portal, we found that a large conceptual gap existed between the ontology-engineering tool and the input (often legacy data).

Management component graphical user interface

The ontology engineer uses the management component to select input data—that is, relevant resources such as HTML and XML documents, DTDs, databases, or existing ontologies that the discovery process can further exploit. Then, using the management component, the engineer chooses from a set of resource-processing methods available in the resource-processing component and from a set of algorithms available in the algorithm library.

The management component also supports the engineer in discovering task-relevant legacy data—for example, an ontology-based crawler gathers HTML documents that are relevant to a given core ontology.

Resource processing

Depending on the available input data, the engineer can choose various strategies for resource processing:

- Index and reduce HTML documents to free text.
- Transform semistructured documents, such as dictionaries, into a predefined relational structure.
- Handle semistructured and structured schema data (such as DTDs, structured database schemata, and existing ontologies) by following different strategies for import, as described later in this article.
- Process free natural text. Our system accesses the natural-language-processing system Saarbrücken Message Extraction System, a shallow-text processor for German.[10] SMES comprises a *tokenizer* based on regular expressions, a *lexical analysis* component including various word *lexicons*, an *amorphological analysis* module, a *named-entity recognizer*, a *part-of-speech tagger,* and a *chunk parser*.

After first preprocessing data according to one of these or similar strategies, the resource-processing module transforms the data into an algorithm-specific relational representation.

Algorithm library

We can describe an ontology by a number of sets of concepts, relations, lexical entries, and links between these entities. We can acquire an existing ontology definition (including L, C, H_C, R, H_R, A, F, and G), using various algorithms that work on this definition and the preprocessed input data. Although specific algorithms can vary greatly from one type of input to the next, a considerable overlap exists for underlying learning approaches such as association rules, formal concept analysis, or clustering. Hence, we can reuse algorithms from the library for acquiring different parts of the ontology definition.

In our implementation, we generally use a multistrategy learning and result combination approach. Thus, each algorithm plugged into the library generates normalized results that adhere to the ontology structures we've discussed and that we can apply toward a coherent ontology definition.

Import and reuse

Given our experiences in medicine,

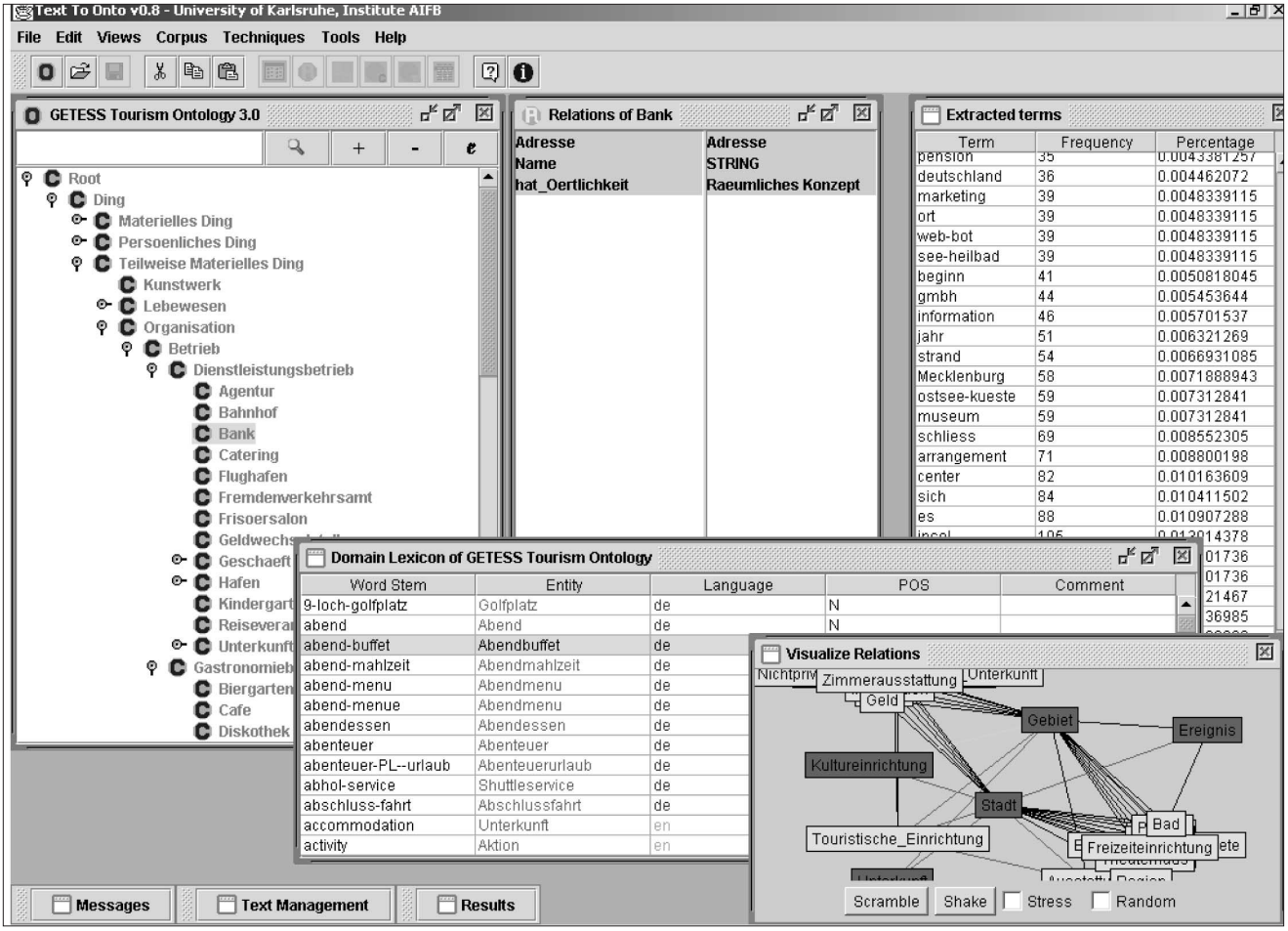


Figure 3. Screenshot of our ontology-learning workbench, Text-To-Onto.

telecommunications, tourism, and insurance, we expect that domain conceptualizations are available for almost any commercially significant domain. Thus, we need mechanisms and strategies to import and reuse domain conceptualizations from existing (schema) structures. We can recover the conceptualizations, for example, from legacy database schemata, DTDs, or from existing ontologies that conceptualize some relevant part of the target ontology.

In the first part of import and reuse, we identify the schema structures and discuss their general content with domain experts. We must import each of these knowledge sources separately. We can also import manually—which can include a manual definition of transformation rules. Alternatively, reverse-engineering tools—such as those that exist for recovering extended entity-relationship diagrams from a given database's SQL description (see the sidebar)—might facilitate the recovery of conceptual structures.

In the second part of the import and reuse step, we must merge or align imported conceptual structures to form a single common ground from which to springboard into the subsequent ontology-learning phases of extracting, pruning, and refining. Although the general research issue of merging and aligning is still an open problem, recent proposals have shown how to improve the manual merging and aligning process. Existing methods mostly rely on matching heuristics for proposing the merger of concepts and similar knowledge base operations. Our research also integrates mechanisms that use an application-data–oriented, bottom-up approach.[11] For instance, formal concept analysis lets us discover patterns between application data and the use of concepts, on one hand, and their heterarchies' relations and semantics, on the other, in a formally concise way (see B. Ganter and R. Wille's work on formal concept analysis in the sidebar).

Overall, the ontology-learning import and reuse step seems to be the hardest to generalize. The task vaguely resembles the general problems encountered in data-warehousing—adding, however, challenging problems of its own.

Extraction

Ontology-extraction models major parts—the complete ontology or large chunks representing a new ontology subdomain—with learning support exploiting various types of Web sources. Ontology-learning techniques partially rely on given ontology parts. Thus, we here encounter an iterative model where previous revisions through the ontology-learning cycle can propel subsequent ones, and more sophisticated algorithms can work on structures that previous, more straightforward algorithms have proposed.

To describe this phase, let's look at some of the techniques and algorithms that we embedded in our framework and implemented in our ontology-learning environment Text-To-Onto (see Figure 3). We cover a substantial part of the overall ontology-learning task in the extraction phase. Text-To-Onto proposes many different ontology learning algorithms for primitives, which we described previously (that is, L, C, R, and so on), to the ontology engineer building on several types of input.

A Common Perspective

Until recently, ontology learning—for comprehensive ontology construction—did not exist. However, much work in numerous disciplines—computational linguistics, information retrieval, machine learning, databases, and software engineering—has researched and practiced techniques that we can use in ontology learning. Hence, we can find techniques and methods relevant for ontology learning referred to as

- "acquisition of selectional restrictions,"[1,2]
- "word sense disambiguation and learning of word senses,"[3]
- "computation of concept lattices from formal contexts,"[4] and
- "reverse engineering in software engineering."[5]

Ontology learning puts many research activities—which focus on different input types but share a common domain conceptualization—into one perspective. The activities in Table A span a variety of communities, with references from 20 completely different events and journals.

References

1. P. Resnik, *Selection and Information: A Class-Based Approach to Lexical Relationships*, PhD thesis, Dept. of Computer Science, Univ. of Pennsylvania, Philadelphia, 1993.
2. R. Basili, M.T. Pazienza, and P. Velardi, "Acquisition of Selectional Patterns in a Sublanguage," *Machine Translation*, vol. 8, no. 1, 1993, pp. 175–201.
3. P. Wiemer-Hastings, A. Graesser, and K. Wiemer-Hastings, "Inferring the Meaning of Verbs from Context," *Proc. 20th Ann. Conf. Cognitive Science Society* (CogSci-98), Lawrence Erlbaum, New York, 1998.
4. B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*, Springer-Verlag, Berlin, 1999.
5. H.A. Mueller et al., "Reverse Engineering: A Roadmap," *Proc. Int'l Conf. Software Eng.* (ICSE-00), ACM Press, New York, 2000, pp. 47–60.
6. P. Buitelaar, *CORELEX Systematic Polysemy and Underspecification*, PhD thesis, Dept. of Computer Science, Brandeis Univ., Waltham, Mass., 1998.
7. H. Assadi, "Construction of a Regional Ontology from Text and Its Use within a Documentary System," *Proc. Int'l Conf. Formal Ontology and Information Systems* (FOIS-98), IOS Press, Amsterdam.
8. D. Faure and C. Nedellec, "A Corpus-Based Conceptual Clustering Method for Verb Frames and Ontology Acquisition," *Proc. LREC-98 Workshop on Adapting Lexical and Corpus Resources to Sublanguages and Applications*, European Language Resources—Distribution Agency, Paris, 1998.
9. F. Esposito et al., "Learning from Parsed Sentences with INTHELEX," *Proc. Learning Language in Logic Workshop* (LLL-2000) and *Learning Language in Logic Workshop* (LLL-2000), Assoc. for Computational Linguistics, New Brunswick, N.J., 2000, pp. 194-198.
10. A. Maedche and S. Staab, "Discovering Conceptual Relations from Text," *Proc. European Conf. Artificial Intelligence* (ECAI-00), IOS Press, Amsterdam, 2000, pp. 321–325.
11. J.-U. Kietz, A. Maedche, and R. Volz, "Semi-Automatic Ontology Acquisition from a Corporate Intranet." *Proc. Learning Language in Logic Workshop* (LLL-2000), ACL, New Brunswick, N.J., 2000, pp. 31–43.
12. E. Morin, "Automatic Acquisition of Semantic Relations between Terms from Technical Corpora," *Proc. of the Fifth Int'l Congress on Terminology and Knowledge Engineering* (TKE-99), TermNet-Verlag, Vienna, 1999.
13. U. Hahn and K. Schnattinger, "Towards Text Knowledge Engineering," *Proc. Am. Assoc. for Artificial Intelligence* (AAAI-98), AAAI/MIT Press, Menlo Park, Calif., 1998.
14. M.A. Hearst, "Automatic Acquisition of Hyponyms from Large Text Corpora," *Proc. Conf. Computational Linguistics* (COLING-92), 1992.
15. Y. Wilks, B. Slator, and L. Guthrie, *Electric Words: Dictionaries, Computers, and Meanings*, MIT Press, Cambridge, Mass., 1996.
16. J. Jannink and G. Wiederhold, "Thesaurus Entry Extraction from an On-Line Dictionary," *Proc. Second Int'l Conf. Information Fusion* (Fusion-99), Omnipress, Wisconsin, 1999.
17. J.-U. Kietz and K. Morik, "A Polynomial Approach to the Constructive Induction of Structural Knowledge," *Machine Learning*, vol. 14, no. 2, 1994, pp. 193–211.
18. S. Schlobach, "Assertional Mining in Description Logics," *Proc. 2000 Int'l Workshop on Description Logics* (DL-2000), 2000; http://SunSITE.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/Vol-33.
19. A. Doan, P. Domingos, and A. Levy, "Learning Source Descriptions for Data Integration," *Proc. Int'l Workshop on The Web and Databases* (WebDB-2000), Springer-Verlag, Berlin, 2000, pp. 60–71.
20. P. Johannesson, "A Method for Transforming Relational Schemas into Conceptual Schemas," *Proc. Int'l Conf. Data Engineering* (IDCE-94), IEEE Press, Piscataway, N.J., 1994, pp. 190–201.
21. Z. Tari et al., "The Reengineering of Relational Databases Based on Key and Data Correlations," *Proc. Seventh Conf. Database Semantics* (DS-7), Chapman & Hall, 1998, pp. 40–52.

Table A. A survey of ontology-learning approaches.

Domain	Methods	Features used	Prime purpose	Papers
Free Text	Clustering	Syntax	Extract	Paul Buitelaar,[6] H. Assadi,[7] and David Faure and Claure Nedellec[8]
	Inductive logic programming	Syntax, logic representation	Extract	Frederique Esposito et al.[9]
	Association rules	Syntax, Tokens	Extract	Alexander Maedche and Steffen Staab[10]
	Frequency-based	Syntax	Prune	Joerg-Uwe Kietz et al.[11]
	Pattern matching	—	Extract	Emanuelle Morin[12]
	Classification	Syntax, semantics	Refine	Udo Hahn and Klemens Schnattinger[13]
Dictionary	Information extraction	Syntax	Extract	Marti Hearst,[14] Yorik Wilks,[15] and Joerg-Uwe Kietz et al.[11]
	Page rank	Tokens	—	Jan Jannink and Gio Wiederhold[16]
Knowledge base	Concept induction, A-Box mining	Relations	Extract	Joerg-Uwe Kietz and Katharina Morik[17] and S. Schlobach[18]
Semistructured schemata	Naive Bayes	Relations	Reverse engineering	Anahai Doan et al.[19]
Relational schemata	Data correlation	Relations	Reverse engineering	Paul Johannesson[20] and Zahir Tari et al.[21]

Lexical entry and concept extraction

One of the baseline methods applied in our framework for acquiring lexical entries with corresponding concepts is lexical entry and concept extraction. Text-To-Onto processes Web documents on the morphological level, including multiword terms such as "database reverse engineering" by n-grams, a simple statistics-based technique. Based on this text preprocessing, we apply term-extraction techniques, which are based on (weighted) statistical frequencies, to propose new lexical entries for L.

Often, the ontology engineer follows the proposal by the lexical entry and concept-extraction mechanism and includes a new lexical entry in the ontology. Because the new lexical entry comes without an associated concept, the ontology engineer must then decide (possibly with help from further processing) whether to introduce a new concept or link the new lexical entry to an existing concept.

Hierarchical concept clustering

Given a lexicon and a set of concepts, one major next step is taxonomic concept classification. One generally applicable method with regard to this is hierarchical clustering, which exploits items' similarities to propose a hierarchy of item categories. We compute the similarity measure on the properties of items.

When extracting a hierarchy from natural-language text, term adjacency or syntactical relationships between terms yield considerable descriptive power to induce the semantic hierarchy of concepts related to these terms.

David Faure and Claure Nedellec give a sophisticated example for hierarchical clustering (see the sidebar). They present a cooperative machine-learning system, Asium (*a*cquisition of semant*i*c knowledge *u*sing *m*achine-learning method), which acquires taxonomic relations and subcategorization frames of verbs based on syntactic input. The Asium system hierarchically clusters nouns based on the verbs to which they are syntactically related and vice versa. Thus, they cooperatively extend the lexicon, the concept set, and the concept heterarchy (L, C, H_C).

Dictionary parsing

Machine-readable dictionaries are frequently available for many domains. Although their internal structure is mostly free text, comparatively few patterns are used to give text definitions. Hence, MRDs exhibit a large degree of regularity that can be exploited to extract a domain conceptualization.

We have used Text-To-Onto to generate a concept taxonomy from an insurance company's MRD (see the sidebar). Likewise, we've applied morphological processing to term extraction from free text—this time, however, complementing several pattern-matching heuristics. Take, for example, the following dictionary entry:

> **Automatic Debit Transfer:** Electronic service arising from a debit authorization of the Yellow Account holder for a recipient to debit bills that fall due direct from the account….

We applied several heuristics to the morphologically analyzed definitions. For instance, one simple heuristic relates the definition term, here *automatic debit transfer*, with the first noun phrase in the definition, here *electronic service*. The heterarchy $H_C : H_C$ (automatic debit transfer, electronic service) links their corresponding concepts. Applying this heuristic iteratively, we can propose large parts of the target ontology—more precisely, L, C, and H_C to the ontology engineer. In fact, because verbs tend to be modeled as relations, we can also use this method to extend R (and the linkage between R and L).

> Targeting completeness for the domain model appears to be practically unmanageable and computationally intractable, but targeting the scarcest model overly limits expressiveness.

Association rules

One typically uses association-rule-learning algorithms for prototypical applications of data mining—for example, finding associations that occur between items such as supermarket products in a set of transactions for example customers' purchases. The generalized association-rule-learning algorithm extends its baseline by aiming at descriptions at the appropriate taxonomy level—for example, "snacks are purchased together with drinks," rather than "chips are purchased with beer," and "peanuts are purchased with soda."

In Text-To-Onto (see the sidebar), we use a modified generalized association-rule-learning algorithm to discover relations between concepts. A given class hierarchy H_C serves as background knowledge. Pairs of syntactically related concepts—for example, pair **(festival,island)** describing the head–modifier relationship contained in the sentence "The festival on Usedom attracts tourists from all over the world."—are given as input to the algorithm. The algorithm generates association rules that compare the relevance of different rules while climbing up or down the taxonomy. The algorithm proposes what appears to be the most relevant binary rules to the ontology engineer for modeling relations into the ontology, thus extending R.

As the algorithm tends to generate a high number of rules, we offer various interaction modes. For example, the ontology engineer can restrict the number of suggested relations by defining so-called restriction concepts that must participate in the extracted relations. The flexible enabling and disabling of taxonomic knowledge for extracting relations is another way of focusing.

Figure 4 shows various views of the results. We can induce a generalized relation from the example data given earlier—**relation rel(event,area)**, which the ontology engineer could name **locatedin**, namely, **events** located in an **area** (which extends L and G). The user can add extracted relations to the ontology by dragging and dropping them. To explore and determine the right aggregation level of adding a relation to the ontology, the user can browse the relation views for extracted properties (see the left side of Figure 4).

Pruning

A common theme of modeling in various disciplines is the balance between completeness and domain-model scarcity. Targeting completeness for the domain model appears to be practically unmanageable and computationally intractable, but targeting the scarcest model overly limits expressiveness. Hence, we aim for a balance between the two that works. Our model should capture a rich target-domain conceptualization but exclude the parts out of its focus. Ontology import and reuse as well as ontology extraction put the scale considerably out of balance where out-of-focus concepts reign. Therefore, we appropriately diminish the ontology in the pruning phase.

We can view the problem of pruning in at least two ways. First, we need to clarify how

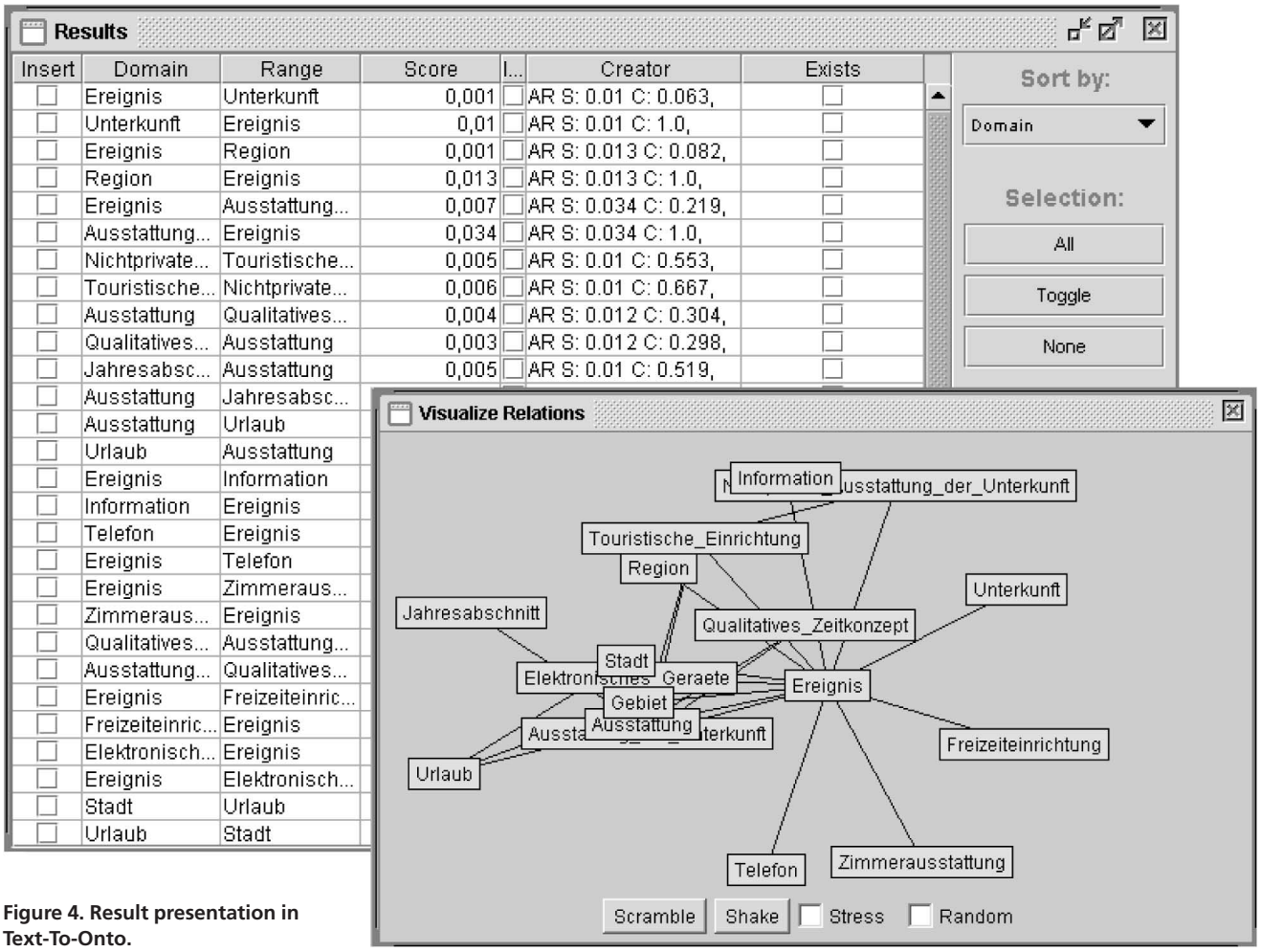


Figure 4. Result presentation in Text-To-Onto.

pruning particular parts of the ontology (for example, removing a concept or relation) affects the rest. For instance, Brian Peterson and his colleagues have described strategies that leave the user with a coherent ontology (that is, no dangling or broken links).[6] Second, we can consider strategies for proposing ontology items that we should either keep or prune. Given a set of application-specific documents, several strategies exist for pruning the ontology that are based on absolute or relative counts of term frequency combined with the ontology's background knowledge (see the sidebar).

Refinement

Refining plays a similar role to extracting—the difference is on a sliding scale rather than a clear-cut distinction. Although extracting serves mainly for cooperative modeling of the overall ontology (or at least of very significant chunks of it), the refinement phase is about fine-tuning the target ontology and the support of its evolving nature. The refinement phase can use data that comes from a concrete Semantic Web application—for example, log files of user queries or generic user data. Adapting and refining the ontology with respect to user requirements plays a major role in the application's acceptance and its further development.

In principle, we can use the same algorithms for extraction and refinement. However, during refinement, we must consider in detail the existing ontology and its existing connections, while extraction works more often than not practically from scratch.

Udo Hahn and Klemens Schnattinger presented a prototypical approach for refinement (see the sidebar)—although not for extraction! They introduced a methodology for automating the maintenance of domain-specific taxonomies. This incrementally updates an ontology as it acquires new concepts from text. The acquisition process is centered on the linguistic and conceptual "quality" of various forms of evidence underlying concept-hypothesis generation and refinement. Particularly, to determine a particular proposal's quality, Hahn and Schnattinger consider semantic conflicts and analogous semantic structures from the knowledge base for the ontology, thus extending an existing ontology with new lexical entries for L, new concepts for C, and new relations for H_C.

Ontology learning could add significant leverage to the Semantic Web because it propels the construction of domain ontologies, which the Semantic Web needs to succeed. We have presented a comprehensive framework for ontology learning that crosses the boundaries of single disciplines, touching on a number of challenges. The good news is, however, that you don't need perfect or optimal support for cooperative ontology modeling. At least according to our experience, cheap methods in an integrated environment can tremendously help the ontology engineer.

While a number of problems remain within individual disciplines, additional challenges arise that specifically pertain to applying ontology learning to the Semantic Web. With the use of XML-based namespace mechanisms, the notion of an ontology with well-defined boundaries—for example, only definitions that are in one file—will disappear. Rather, the Semantic Web might yield an amoeba-like structure regarding ontology boundaries because ontologies refer to and import each other (for example, the DAML-ONT primitive **import**). However, we do not yet know what the semantics of these structures will look like. In light of these facts, the importance of methods such as ontology pruning and crawling will drastically increase. Moreover, we have so far restricted our attention in ontology learning to the conceptual structures that are almost contained in RDF(S). Additional semantic layers on top of RDF (for example, future OIL or DAML-ONT with axioms, *A*) will require new means for improved ontology engineering with axioms, too! ■

Acknowledgments

We thank our students, Dirk Wenke, and Raphael Volz for work on *OntoEdit* and *Text-To-Onto*. Swiss Life/Rentenanstalt (Zurich, Switzerland), Ontoprise GmbH (Karlsruhe, Germany), the US Air Force DARPA DAML ("OntoAgents" project), the European Union (IST-1999-10132 "On-To-Knowledge" project), and the German BMBF (01IN901C0 "GETESS" project) partly financed research for this article.

References

1. E. Grosso et al., "Knowledge Modeling at the Millennium—the Design and Evolution of Protégé-2000," *Proc. 12th Int'l Workshop Knowledge Acquisition, Modeling and Management* (KAW-99), 1999.
2. G. Webb, J. Wells, and Z. Zheng, "An Experimental Evaluation of Integrating Machine Learning with Knowledge Acquisition," *Machine Learning*, vol. 35, no. 1, 1999, pp. 5–23.
3. K. Morik et al., *Knowledge Acquisition and Machine Learning: Theory, Methods, and Applications*, Academic Press, London, 1993.
4. B. Gaines and M. Shaw, "Integrated Knowledge Acquisition Architectures," *J. Intelligent Information Systems*, vol. 1, no. 1, 1992, pp. 9–34.
5. K. Morik, "Balanced Cooperative Modeling," *Machine Learning*, vol. 11, no. 1, 1993, pp. 217–235.
6. B. Peterson, W. Andersen, and J. Engel, "Knowledge Bus: Generating Application-Focused Databases from Large Ontologies," *Proc. Fifth Workshop Knowledge Representation Meets Databases* (KRDB-98), 1998, http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-10 (current 19 Mar. 2001).
7. S. Staab et al., "Knowledge Processes and Ontologies," *IEEE Intelligent Systems*, vol. 16, no. 1, Jan./Feb. 2001, pp. 26–34.
8. S. Staab and A. Maedche, "Knowledge Portals—Ontologies at Work," to be published in *AI Magazine*, vol. 21, no. 2, Summer 2001.
9. G. Miller, "WordNet: A Lexical Database for English," *Comm. ACM*, vol. 38, no. 11, Nov. 1995, pp. 39–41.
10. G. Neumann et al., "An Information Extraction Core System for Real World German Text Processing," *Proc. Fifth Conf. Applied Natural Language Processing* (ANLP-97), 1997, pp. 208–215.
11. G. Stumme and A. Maedche, "FCA-Merge: A Bottom-Up Approach for Merging Ontologies," to be published in *Proc. 17th Int'l Joint Conf. Artificial Intelligence* (IJCAI '01), Morgan Kaufmann, San Francisco, 2001.

The Authors

Alexander Maedche is a PhD student at the Institute of Applied Informatics and Formal Description Methods at the University of Karlsruhe. His research interests include knowledge discovery in data and text, ontology engineering, learning and application of ontologies, and the Semantic Web. He recently founded together with Rudi Studer a research group at the FZI Research Center for Information Technologies at the University of Karlsruhe that researches Semantic Web technologies and applies them to knowledge management applications in practice. He received a diploma in industrial engineering, majoring in computer science and operations research, from the University of Karlsruhe. Contact him at the Institute AIFB, Univ. of Karlsruhe, 76128 Karlsruhe, Germany; ama@aifb.uni-karlsruhe.de.

Steffen Staab is an assistant professor at the University of Karlsruhe and cofounder of Ontoprise GmbH. His research interests include computational linguistics, text mining, knowledge management, ontologies, and the Semantic Web. He received an MSE from the University of Pennsylvania and a Dr. rer. nat. from the University of Freiburg, both in informatics. He organized several national and international conferences and workshops, and is now chairing the Semantic Web Workshop in Hongkong at WWW10. Contact him at the Institute AIFB, Univ. of Karlsruhe, 76128 Karlsruhe, Germany; sst@aifb.uni-karlsruhe.de.