ABSTRACT

Title of Dissertation: TOWARDS THE SEMANTIC WEB: KNOWLEDGE REPRESENTATION IN A DYNAMIC, DISTRIBUTED ENVIRONMENT

Jeffrey Douglas Heflin, Doctor of Philosophy, 2001

Dissertation directed by: Professor James A. Hendler
Department of Computer Science

The World Wide Web is an information resource with virtually unlimited potential. However, this potential is relatively untapped because it is difficult for machines to process and integrate this information meaningfully. Recently, researchers have begun to explore the potential of associating web content with explicit meaning, in order to create a Semantic Web. Rather than rely on natural language processing to extract this meaning from existing documents, this approach requires authors to describe documents using a knowledge representation language.

Although knowledge representation can solve many of the Web's problems, existing research cannot be directly applied to the Semantic Web. Unlike most traditional knowledge bases, the Web is highly decentralized, changes rapidly, and contains a staggering amount of information. This thesis examines how knowledge

representation must change to accommodate these factors. It presents a new method for integrating web data sources based on ontologies, where the sources explicitly commit to one or more autonomously developed ontologies. In addition to specifying the semantics of a set of terms, the ontologies can extend or revise one another. This technique permits automatic integration of sources that commit to ontologies with a common descendant, and when appropriate, of sources that commit to different versions of the same ontology.

The potential of the Semantic Web is demonstrated using SHOE, a prototype ontology language for the Web. SHOE is used to develop extensible shared ontologies and create assertions that commit to particular ontologies. SHOE can be reduced to datalog, allowing it to scale to the extent allowed by the optimized algorithms developed for deductive databases. To demonstrate the feasibility of the SHOE approach, we describe a basic architecture for a SHOE system and a suite of general purpose tools that allow SHOE to be created, discovered, and queried. Additionally, we examine the potential uses and difficulties associated with the SHOE approach by applying it to two problems in different domains.

TOWARDS THE SEMANTIC WEB:
KNOWLEDGE REPRESENTATION IN A
DYNAMIC, DISTRIBUTED ENVIRONMENT

by

Jeffrey Douglas Heflin

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2001

Advisory Committee:

Professor James A. Hendler, Chairman/Advisor
Professor Ashok Agrawala
Assistant Professor Benjamin Bederson
Professor Dana Nau
Professor Dagobert Soergel
Professor V.S. Subrahmanian

ACKNOWLEDGEMENTS

I would like to thank my advisor, James Hendler, for his guidance throughout my years at Maryland. He has prepared me well for a career as a computer science academic and has worked hard to promote my work. I would also like to thank Ashok Agrawala, Benjamin Bederson, Dana Nau, Dagobert Soergel, and V.S. Subrahmanian for being on my defense committee and for their useful feedback and questions. Additionally, I thank Professor Don Perlis for his comments on Chapter 3 of this thesis.

The PLUS Group was a stimulating place to work, and I want to thank both its present and former members. I especially thank Sean Luke for developing the original idea for the SHOE language and for his help as I expanded upon the foundation that he laid. I also want to thank him for choosing to pursue another research topic, allowing me to become the chief "SHOE guy." Additionally, I thank Carolyn Gasarch for her help with both Parka and the SHOE Search tool, Qin Zhendong for his help with the Knowledge Annotator, and Merwyn Taylor for fixing Parka bugs and making much needed enhancements. Although his work was orthogonal to SHOE, I also thank Jaime Montemayor for our discussions about AI in general and each other's research.

While working towards my degree, I was also a part-time employee of ISX Corporation. I thank them for providing me with an opportunity to explore the practical application of SHOE to the needs of their client, and providing me with a much needed supplement to the meager income that a graduate student normally has. In particular, I'd like to thank my project manager Brian Kettler and technical lead Terry Padgett for thought-provoking questions and discussions.

Finally, I must thank my family, who have provided love and support over all of these years. My mother and father have always believed in me, which has helped me get to where I am now. My brother Randy has been a great friend, and his enthusiasm for his work as a physicist helped to convince me that a career in academia might be right for me as well. Most importantly, I want to thank my beautiful wife Katherine for her love and understanding. She never complained when I had to spend a weekend working on this dissertation, and always offered words of encouragement when I was feeling down.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The World Wide Web is the greatest repository of information ever assembled by man. It contains documents and multimedia resources concerning almost every imaginable subject, and all of this data is instantaneously available to anyone with an Internet connection. The Web's success is largely due to its decentralized design: web pages are hosted by numerous computers, where each document can point to other documents, either on the same or different computers. As a result, individuals all over the world can provide content on the Web, allowing it to grow exponentially as more and more people learn how to use it.

However, the Web's size has also become its curse. Due to the sheer volume of available information, it is becoming increasingly difficult to locate useful information. Although directories (such as Yahoo!) and search engines (such as Google and Alta Vista) can provide some assistance, they are far from perfect. For many users, locating the "right" document is still like trying to find a needle in a haystack.

Furthermore, users often want to use the Web to do more than just locate a document, they want to perform some task. For example, a user might want to find the best price on a desktop computer, plan and book a romantic vacation to a

Caribbean island, or make reservations at a moderately-priced Italian restaurant within five blocks of the movie they plan to see that evening. Completing these tasks often involves visiting a series of pages, integrating their content and reasoning about them in some way. This is far beyond the capabilities of contemporary directories and search engines, but could they eventually perform these tasks?

The main obstacle is the fact that the Web was not designed to be processed by machines. Although, web pages include special information that tells a computer how to display a particular piece of text or where to go when a link is clicked, they do not provide any information that helps the machine to determine what the text means. Thus, to process a web page intelligently, a computer must understand the text, but natural language understanding is known to be an extremely difficult and unsolved problem.

Some researchers and web developers have proposed that we augment the Web with languages that make the meaning of web pages explicit. Tim Berners-Lee, inventor of the Web, has coined the term *Semantic Web* to describe this approach. Berners-Lee, Hendler and Lassila [4] provide the following definition:

> The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.

Before we delve more deeply into just what the Semantic Web is, we will examine some of the problems that it is meant to solve.

1.1 Why Search is Lacking

Users have two main tools to help them locate relevant resources on the Web, catalogs and search engines. Catalogs are constructed by human experts, thus they tend to be highly accurate but can be difficult to maintain as the Web grows. To keep up with this growth, search engines were designed to eliminate human effort in cataloging web sites. A search engine consists of a mechanism that "crawls" the Web looking for new or changed pages, an indexing mechanism, and a query interface. Typically, the indices store information on the frequency of words and some limited positional information. Users query the system by entering a few keywords and the system computes its response by matching the entries against the index. Although many contemporary search engines now also use link analysis to some degree, this only helps to identify the most popular pages, which may or may not be related to the relevance of the pages for a particular query.

Although search engines are able to index large portions of the Web, users often experience one of two problems: they either get back too many irrelevant results or no results at all. The first problem arises because the same word can have different meanings in different contexts and keyword indices do not preserve the notion of relationships between words. Although rarer, the second problem is due to the use of a term (or set of terms) that does not appear in the web pages. Although queries can sometimes be improved by either adding more specific words to the query (in the former case) or synonyms (in the latter case), many useful queries are still beyond the capabilities of contemporary search engines.

Let us use an example to illustrate some of the problems with contemporary web search. Consider a query to find the chair of MIT's computer science department. A reasonable set of search terms might be "MIT computer science chair." As shown in

Figure 1.1, even Google, one of the most acclaimed search engines does not return the desired information. [1]

Why was this query unsuccessful? In the first result, the string “MIT” is matched to the German word “mit.” As a result, information from a German computer science department is returned. The second result is for the Committee on the Status of Women in Computing Research. This page talks about the chairs of the committee, who are both members of computer science departments, and lists members, one of which is affiliated with MIT. Thus, the problem is that the search engine is not aware of the desired relationship between the terms “chair,” “MIT,” and “computer science.” Similarly, the third result is a person who received their degree from MIT, is a professor of computer science at another institution, and was chair of another organization.

The lack of an ability to understand the context of words and relationships between search terms explains many of the false positives identified by the search engine, but why was the desired document missed? As it turns out, instead of a chair, MIT has a department head, which is a similar but not identical notion. Our lack of knowledge about the department prevented us from providing the search engine with a query that would allow it to find the correct result. However, if we had asked a person the same query, they might assume that we meant the the head of MIT’s computer science department, or at least ask us to clarify our intention. If a search engine could understand the intended meaning of the words, or even better, the semantic relationships between them, then more accurate searches would be possible. This is one of the goals of the Semantic Web.

[1] Since search engines are constantly changing, you may get different results if you try this query yourself.

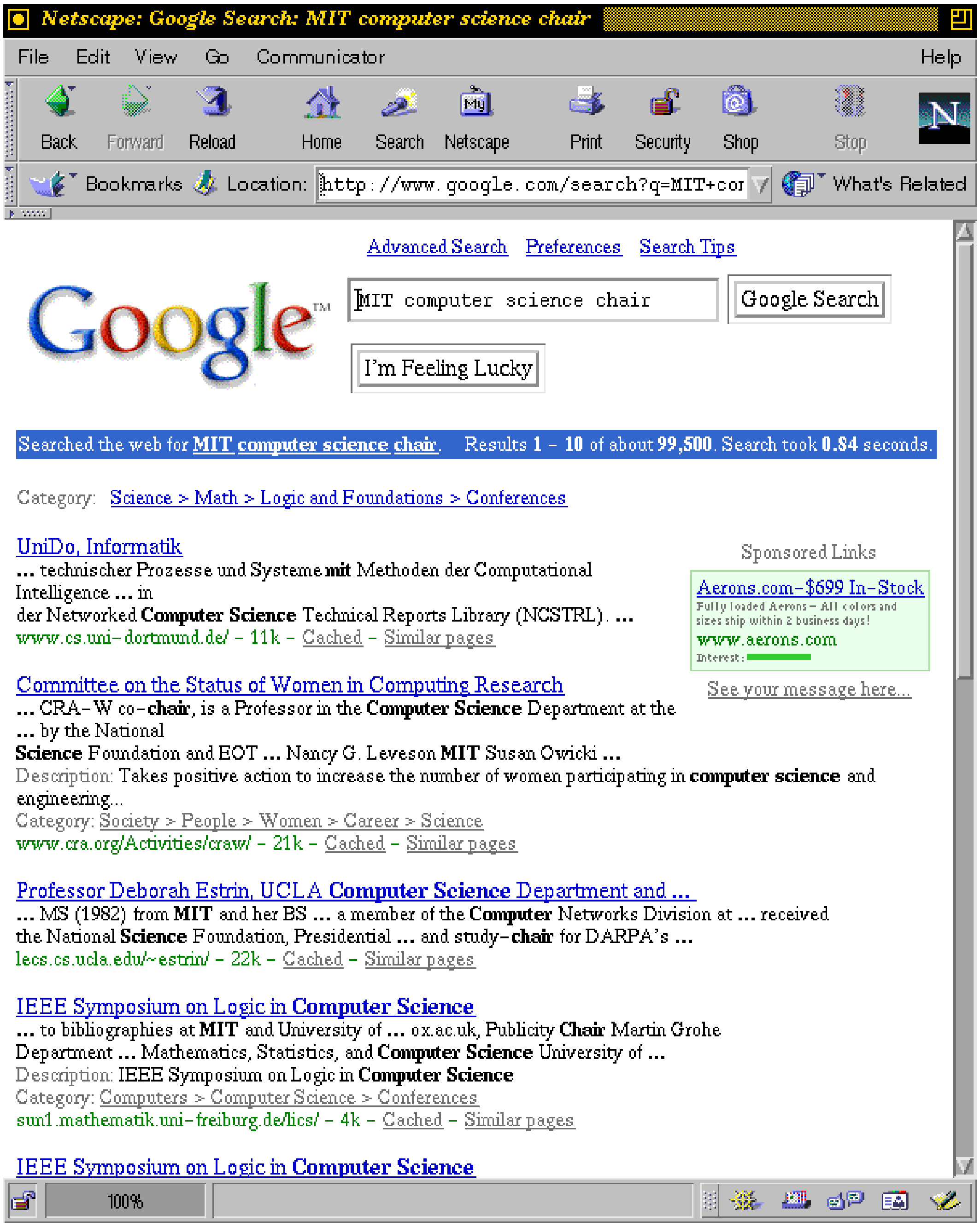

Figure 1.1: Results of query to find the chair of MIT's computer science department.

1.2 Other Applications of the Semantic Web

Improved search is only one of the many potential benefits from a Semantic Web. Internet agents, which are autonomous programs that interact with the Internet, can also benefit. In order to accomplish some goal, an internet agent can request and perceive web pages, and execute web services. Theoretically, such agents are capable of comparison shopping, participating in an auction, or arranging a complete vacation. For example, an agent might be asked to make reservations for a trip to Jamaica, and the agent would book a flight, arrange for a rental car, and reserve a hotel room, all based on the cheapest rates available. Although there are already agents that can perform some of these tasks, they are built to handle only a predefined set of web pages and are highly dependent on the structure of these pages. Thus, they are very brittle; if a web page changes, the agent may no longer be able to locate information or interact with it. Agents that could consider the semantics of a web page instead of its layout would be much more robust.

The Semantic Web could also make push systems more practical. A push system changes the way that users are connected with data: instead of forcing information consumers to find relevant web pages, the web pages are instead "pushed" to them. Such systems require a profile of the user and a method to evaluate if a web page is relevant to a given profile. However, unless this evaluation method is very accurate, the system will keep "pushing" unwanted information to the user, reducing its utility. Such results will tend more to annoy than to help the user, and are usually turned off. Web pages with semantic content can be evaluated and pushed more accurately.

Finally, the Semantic Web may allow users to organize and browse the Web in ways more suitable to the problems they have at hand. The Semantic Web could be used to impose a conceptual filter to a set of web pages, and display their relationships based on such a filter. This may also allow visualization of complex content. With HTML, such interfaces are virtually impossible since it is difficult to extract meaning from the text.

1.3 Knowledge Representation on the Web

The Semantic Web depends on the ability to associate formal meaning with content. The field of knowledge representation (discussed in Section 2.2 provides a good starting point for the design of a Semantic Web language because it offers insight into the design and use of languages that attempt to formalize meaning. However, the nature of the Web challenges many of the assumptions of traditional knowledge representation work, and requires us to look at the problem from a new perspective. The impact of some of the most significant characteristics of the Web are discussed below:

- **The Web is distributed.** One of the driving factors in the proliferation of the Web is the freedom from a centralized authority. However, since the Web is the product of many individuals, the lack of central control presents many challenges for reasoning with its information. First, different communities will use different vocabularies, resulting in problems of synonymy (when two different words have the same meaning) and polysemy (when the same word is used with different meanings). Second, the lack of editorial review or quality control means that each page's reliability must be questioned. An intelligent web agent simply cannot assume that all of the information it gathers is correct and consistent. There are quite a number of well-known "web hoaxes" where information was published on the Web with the intent to amuse or mislead.

Furthermore, since there can be no global enforcement of integrity constraints on the Web, information from different sources may be in conflict. Some of these conflicts may be due to philosophical disagreement; different political groups, religious groups, or nationalities may have fundamental differences in opinion that will never be resolved. Any attempt to prevent such inconsistencies must favor one opinion, but the correctness of the opinion is very much in the "eye of the beholder."

- **The Web is dynamic.** The web changes at an incredible pace, much faster than a user or even an intelligent web agent can keep up with. While new pages are being added, the content of existing pages is changing. Some pages are fairly static, others change on a regular basis and still others change at unpredictable intervals. These changes may vary in significance: although the addition of punctuation, correction of spelling errors, or reordering of a paragraph does not affect the semantic content of a document; other changes may completely alter meaning, or even remove large amounts of information. A web agent must assume that its data can be, and often will be, out of date.

  The rapid pace of information change on the Internet poses an additional challenge to any attempt to create standard vocabularies and provide formal semantics. As understanding of a given domain changes, both the vocabulary may change and the semantics may be refined. It is important that such changes do not adversely alter the meaning of existing content.

- **The Web is massive.** In the year 2000, estimates placed the number of indexable web pages at over 2 billion, and predicted that this number would double in 2001. Even if each page contained only a single piece of agent-gatherable knowledge, the cumulative database would be large enough to bring most reasoning systems to their knees. To scale to the size of the ever growing Web, we must either restrict the expressivity of our representation language or use incomplete reasoning algorithms.

- **The Web is an open world.** A web agent is not free to assume it has gathered all available knowledge; in fact, in most cases an agent should assume it has gathered rather little available knowledge. Even the largest search engines have only crawled about 25% of the available pages. However, in order to deduce more facts, many reasoning systems make the closed-world assumption. That is, they assume that anything not entailed in the knowledge base is not true. Yet it is clear that the size and evolving nature of the Web makes it unlikely that any knowledge base attempting to describe it could ever be complete.

In this thesis, we will analyze these problems, provide a method for integrating web data, and introduce SHOE, a prototype language that demonstrates the potential of the Semantic Web.

1.4 Contributions

In this dissertation, I will describe three significant contributions I have made to the field of knowledge representation as it applies to the Semantic Web:

- I provide a new formal definition of ontologies for use in dynamic, distributed environments, such as the World Wide Web. In addition to specifying the semantics of a set of terms, ontologies are objects that can extend or revise one another. An ontology can also specify compatibility with earlier versions of itself.

- I develop a new method for integrating distributed data sources. In this method, sources explicitly commit to one or more autonomously developed ontologies. This technique permits automatic integration of sources that commit to ontologies with a common descendant, and when appropriate, of sources that commit to different versions of the same ontology.

- I introduce SHOE, a web-based knowledge representation language that allows machines to automatically process and integrate web data. Authors provide data in an XML format and explicitly commit to one or more shared ontologies that provide semantics for the terms. Using the method described above, SHOE can automatically integrate web data, even when ontologies evolve independently from the data sources that commit to them.

Additionally, I demonstrate the feasibility and potential use of SHOE as a semantic web language in two ways:

- I designed and implemented a basic architecture for semantic web systems. The implementation includes two reusable libraries, and four general purpose SHOE tools, totaling more than 20,000 lines of code. Included in this suite are tools to help users create SHOE documents, a web-crawler that gathers information from pages and stores it in a repository, and a powerful user interface for querying the repository.

- I developed and deployed two applications to show how SHOE can be used in practice. For the first application, I used a variety of techniques to rapidly create nearly 40,000 SHOE assertions from the web pages of 15 different computer science departments, and then deployed a new means for searching these pages. For the second application, I worked with a team of doctors and scientists to create a food safety ontology with over 150 categories and relations, and developed a special tool that uses SHOE assertions to solve an important problem in this domain.

1.5 Thesis Overview

The next chapter of this thesis surveys work from a number of related fields. The Semantic Web is an emerging research area, but builds on existing technologies for the World Wide Web and the literature of knowledge representation. Additionally, deductive databases provide algorithms and insights for using logic in large-data situations such as the Web and the work in distributed databases has considered the problem of managing information in decentralized environments. In addition to introducing the reader to these diverse areas, the chapter discusses their relevance to the topic of the thesis.

Chapter 3 formally examines the problems of the Semantic Web and describes various methods for integrating data in this unique environment. The approach begins with a first-order logic language, and adds various notions of ontologies to cope with the problem of integration of autonomous sources that commit to different ontologies or different versions of ontologies. This may be skipped by those who are only interested in the pragmatics of the Semantic Web, but is essential for anyone who is designing a Semantic Web language.

The next three chapters describe the design and use of SHOE, the first ontology-based semantic web language. Chapter 4 describes the syntax and semantics of the language. These semantics are grounded in the framework presented in Chapter 3. Chapter 5 discusses the issues of implementing the SHOE language and provides a basic system architecture. It then describes a number of specific tools that have been developed to demonstrate the use of SHOE. Chapter 6 discusses the practical issues of using the language by describing two case studies. These examples demonstrate how SHOE can be used by both general search systems and special purpose query tools.

The remainder of the thesis is a comparison to other languages and the conclusions. Chapter 7 compares SHOE to the other leading Semantic Web languages, which include Ontobroker, RDF, and OIL. It also describes DAML+OIL, an international effort to combine the best features of these languages. Chapter 8 provides an analysis of the language and discusses future directions of the Semantic Web. Appendix A provides both SGML and XML DTDs that precisely define the grammar of the SHOE language.

Chapter 2

Background

The Semantic Web is an emerging research area which builds on the foundations of diverse prior work. First, since the Semantic Web will be built on top of the existing Web, it is important to have a clear understanding of existing Web standards, and to anticipate how the Semantic Web will interact with other Web technologies. Second, the field of knowledge representation is directly concerned with the issue of semantics, and has resulted in many languages from which ideas can be drawn. However, scalability is a problem for many traditional KR systems. Thus, work in deductive databases, which has studied reasoning with large amounts of data, may help us design inference algorithms that can scale to the size of the Web. Finally, it is possible to view the Web as a collection of autonomous databases. Thus work on distributed databases, particularly in the area of semantic heterogeneity, is highly relevant. In this chapter we will discuss each of these areas and how they relate to the Semantic Web. We leave the discussion of various Semantic Web languages until Chapter 7.

2.1 The World Wide Web

In order to understand the World Wide Web, we will first look at the Internet, the infrastructure upon which it was built. Then we will examine the Hypertext Markup Language (HTML), which is the language used to describe the majority of existing web pages. Finally, we will discuss the eXtensible Markup Language (XML), which may serve as a foundation for the Semantic Web.

2.1.1 The Internet

The Internet's roots begin with ARPANET, a project commissioned by the Advanced Research Projects Agency (ARPA) to study country-wide data communication. In 1969, ARPANET consisted of four computers (called hosts) in different cities, but connected by a network. ARPANET grew over the years and electronic mail became a popular early application. In 1973, ARPA introduced the "Internetworking" program, with the goal of developing an open architecture network, where different networks might have different architectures, but could interwork via a meta-level "Internetworking Architecture." A new network protocol was needed to support this architecture, which led to the creation of the Transmission Control Protocol (TCP) and Internet Protocol (IP), jointly known as (TCP/IP). TCP/IP is the low-level protocol used by most traffic on the Internet today. High level protocols such as the File Transport Protocol (FTP), TELNET, the Simple Mail Transfer Protocol (SMTP), and the Hypertext Transfer Protocol (HTTP), all rely on TCP/IP in order to transfer files, perform remote logins, transfer electronic mail, and exchange Web documents using the Internet.

2.1.2 Development of the Web

In 1990, Tim Berners-Lee developed the first version of his World Wide Web program at CERN. The concept behind Berners-Lee's invention was to use hypertext as a means of organizing a distributed document system. Hypertext refers to a collection of documents with cross-references (also known as links) between them that enable readers to peruse the text in a nonsequential manner. In order to make the Web work on the Internet, Berners-Lee had to develop a mechanism for addressing documents on different machines, a protocol that allowed computers to request documents, and a simple language to describe the documents.

The mechanism for addressing objects is the Uniform Resource Locator (URL). A URL consists of a scheme followed by a colon and a scheme-specific part. The scheme specifies a protocol by which the object is accessed and determines the form of the scheme-specific part. The most commonly used scheme is http, in which the scheme-specific part consists of the name of the host machine, pathname of a file, and an optional reference to an anchor in the file. Sometimes, the term URL is used interchangeably with the term Uniform Resource Identifier (URI), although technically URI is a broader term used to indicate strings that may identify web resources without specifying the primary access mechanism. Many URI schemes, including the http one, set up hierarchical namespaces. These schemes often use the Domain Name System (DNS) to identify the authority for the namespace. DNS, which is a hierarchical namespace itself, has been successfully used to name hosts on the Internet. The use of hierarchical namespaces allow new URIs to be created without need for approval by a single central authority, while guaranteeing that URIs created by different authorities are distinct.

The Hypertext Transport Protocol (HTTP) is used to request documents. These

```
<HTML>
  <HEAD>
    <TITLE>Acme CD Store</TITLE>
  </HEAD>

  <BODY>
    <H1>Acme CD Store</H1>
    <P>Welcome to our CD store!</P>

    <H2>Catalog</H2>
    <UL>
      <LI>1. Cracker - Kerrosene Hat: $15.99
      <LI>2. Phair, Liz - Exile in Guyville: $15.99
      <LI>3. Soul Coughing - Irresistible Bliss: $15.99
      ...
    </UL>

    <P><A HREF="order.html">Place your order!</A></P>
  </BODY>
</HTML>
```

Figure 2.1: An example HTML document.

requests are received by programs called web servers that run on the host machine. A web server uses a URL provided in the request to determine which file to deliver. Often, this file is a Hypertext Markup Language (HTML) document. A program on the requesting machine called a browser renders the HTML for presentation on the screen.

HTML is essentially a text stream with special codes embedded. These codes, called tags, are identified by having angle-brackets that surround them. An example HTML document is given in Figure 2.1. The most important tag in HTML is the anchor tag, indicated by <A>. With the <A HREF=...> form, anchor tags create a hypertext link to another document by specifying a URL. These tags indicate that a web browser should retrieve the document represented by the URL if the link is activated. Most of HTML's other tags were concerned with organization and presentation of the document. The first version of the language included tags to indicate headings (<H1>, <H2>, etc.), paragraphs (<P>), and lists (<UL> and <LI>). Later versions added tags for special formatting such as bold and italics, forms that allowed pages to be interactive, tables, and text-flow around images.

Although HTML's tags are mostly presentation oriented, some tags were added to provide weak semantic information. HTML 2.0 [3] introduced the META element and the REL attribute. The META element specifies meta-data in the form of a name/value pair. A popular use for META was to indicate keywords, for example <META name="keywords" content="Semantic Web">, that could help search engines index the site. However, many sites began abusing the keywords by including popular keywords that did not accurately describe the site (this is known as keyword spamming). As a result, many search engines now ignore this tag. The REL attribute of the anchor (<A>) and link (<LINK>) elements names a relationship from the enclosing document to the document pointed to by a hyperlink; the REV attribute names the relationship in the reverse direction. HTML 3.0 [80] added the CLASS attribute, which could be used within almost any tag to create semantic subclasses of that element, Unfortunately, the semantic markup elements of HTML are rarely used, but even if they were, the semantics they provide is limited.

To address the semantic limitations of HTML, Dobson and Burrill [24] attempted to reconcile it with the Entity-Relationship (ER) database model. This is done by supplementing HTML with a simple set of tags that define "entities" within documents, labeling sections of the body text as "attributes" of these entities, and defining relationships from an entity to outside entities. This was the first attempt to formally add structured data to web pages and thus presented an approach to a problem that was also a significant motivation behind the design of XML.

2.1.3 XML

Despite its popularity, HTML suffered from two problems. First, whenever someone felt that HTML was insufficient for their needs, they would simply add additional tags to their documents, resulting in a number of non-standard variants. Second, because HTML was mostly designed for presentation to humans, it was difficult for machines to extract content and perform automated processing on the documents. To solve these problems, the World Wide Web Consortium (W3C) developed the Extensible Markup Language (XML) [15].

XML is essentially a subset of the Standard Generalized Markup Language (SGML) [35], a standard used by the text processing community. SGML is a meta-language, in the sense that it can be used to define other languages, called SGML applications. The benefits of SGML include platform independence, separation of content from format, and the ability to determine if documents conform to structural rules. XML kept these features, but left out those that were infrequently used, confusing, or difficult to implement.

XML's syntax will seem familiar to users of HTML. This is not surprising, since HTML is an application of SGML, XML's parent language. Like HTML (and SGML), XML allows angle-bracketed tags to be embedded in a text data stream, and these tags provide additional information about the text. However, unlike HTML, XML does not provide any meaning for these tags. Thus, the tag `<P>` may mean paragraph, but it may mean part instead. An XML version of our CD store example is given in Figure 2.2.

```xml
<?xml version="1.0"?>
<catalog>
   <cd>
      <artist>Cracker</artist>
      <title>Kerosene Hat</title>
      <price currency="USD">15.99</price>
   </cd>
   <cd>
      <artist>Phair, Liz</artist>
      <title>Exile in Guyville</title>
      <price currency="USD">15.99</price>
   </cd>
   <cd>
      <artist>Soul Coughing</artist>
      <title>Irresistible Bliss</title>
      <price currency="USD">15.99</price>
   </cd>
</catalog>
```

Figure 2.2: An example XML document.

There are three kinds of tags in XML: start tags, end tags, and empty-element tags. A start tag consists of a name and a set of optional attributes, surrounded by angle-brackets. Each attribute is a name/value pair, separated by an equal sign. In the example, each price tag has a currency attribute. An end tag consists of the name from a previous start tag, but preceded by a slash (“/”) and cannot have any attributes. Every start tag must have exactly one matching end tag. Empty-element tags are like start tags, but don’t have a matching end tag. Instead, an empty element is indicated by a slash just before the closing bracket. For example, <IMG SRC=”photo.jpg” /> would be an empty-element tag.

The data from a start tag to an end tag comprises an *element*. An element can contain other elements, free text, or a combination of the two between its start and end tags. A well-formed XML document contains exactly one top-level element, but can have an arbitrary nesting of elements within that element.

Although XML’s flexibility makes it easy for authors to describe arbitrary content quickly and easily, this flexibility can be problematic for machine processing. Since XML cannot express the meaning of tags, most processing applications require tag sets whose meanings have been agreed to by some standard or convention. To help with machine processing, XML allows grammars to be defined for XML tags. This information is contained in a document type definition (DTD) that specifies valid elements, the contents of these elements, and which attributes may modify an element. We will not discuss the details of DTDs, which can be quite complicated, but suffice to say that they essentially define a context free grammar. An XML document that has an associated DTD and conforms to the rules defined in it is said to be *valid*.

Although a DTD provides a syntax for an XML document, the semantics of a DTD are implicit. That is, the meaning of an element in a DTD is either inferred by a human due to the name assigned to it, is described in a natural-language comment within the DTD, or is described in a document separate from the DTD. Humans can then build these semantics into tools that are used to interpret or translate the XML documents, but software tools cannot acquire these semantics independently. Thus, an exchange of XML documents works well if the parties involved have agreed to a DTD beforehand, but becomes problematic when one wants to search across a set of DTDs or to spontaneously integrate information from multiple sources.

One of the hardest problems in any integration effort is mapping between different representations of the same concepts – the problem of integrating DTDs is no different. One difficulty is identifying and mapping differences in naming conventions. As with natural language, XML DTDs have the problems of polysemy and synonymy. For example, the elements <PERSON> and <INDIVIDUAL> might be synonymous. Similarly, an element such as <SPIDER> might be polysemous: in one document it could mean a piece of software that crawls the World Wide Web while in another it means an arachnid that crawls a web of the silky kind. Furthermore, naming problems can apply to attribute names just as easily as they apply to element names. In general, machines do not have access to the contextual information that humans have, and thus even an automated dictionary or thesaurus would be of little help in resolving the problems with names described here.

An even more difficult problem is identifying and mapping differences in structure. XML's flexibility gives DTD authors a number of choices. Designers attempting to describe the same concepts may choose to do so in many different ways. In Figure 2.3, three possible representations of a person's name are shown. One choice involves whether the name is a string or is an element with structure of its own. Another choice is whether the name is an attribute or an element. One of the

```
<!-- The NAME is a subelement with character content -->
<PERSON>
   <NAME>John Smith</NAME>
</PERSON>

<!-- The NAME is a subelement with element content -->
<PERSON>
   <NAME><FNAME>John</FNAME><LNAME>Smith</LNAME></NAME>
</PERSON>

<!-- The NAME is an attribute of PERSON -->
<PERSON NAME="John Smith">
```

Figure 2.3: Structural differences in XML representation.

reasons for these problems is the lack of semantics in XML. There is no special meaning associated with attributes or content elements. Element content might be used to describe properties of an object or group related items, while attributes might be used to specify supplemental information or single-valued properties.

Once humans have identified the appropriate mappings between two DTDs, it is possible to write XSL Transformations (XSLT) stylesheets [18] that can be used to automatically translate one document into the format of another. Although this is a good solution to the integration problem when only a few DTDs are relevant, it is unsatisfactory when there are many DTDs; if there are n DTDs, then there would need to be O(n^2) different stylesheets to allow automatic transformation between any pair of them. Furthermore, when a DTD was created or revised, someone would have to create or revise the n stylesheets to transform it to all other DTDs. Obviously, this is not a feasible solution.

Of course, the problems of mapping DTDs would go away if we could agree on a single universal DTD, but even at the scale of a single corporation, data standardization can be difficult and time consuming – data standardization on a worldwide scale would be impossible. Even if a comprehensive, universal DTD was possible, it would be so unimaginably large that it would be unusable, and the size of the standards committee that managed it would preclude the possibility of extension and revision at the pace required for modern data processing needs.

Recently, the W3C has released an alternative to DTDs called XML Schema [85, 6]. XML Schemas provide greater flexibility in the definition of an XML application, even allowing the definition of complex data types. Furthermore, XML Schemas use the same syntactic style as other XML documents. However, XML Schema only gives XML an advanced grammar specification and datatyping capability, and still suffers from the same semantic drawbacks as DTDs.

The lack of semantics in XML DTDs and XML Schemas makes it difficult to integrate XML documents. In the next section, we discuss the field of knowledge representation, which has been concerned with semantic issues. In Section 7.2, we will discuss the Resource Description Framework (RDF), a W3C standard that attempts to address some of the semantic problems of XML.

2.2 Knowledge Representation

Many of the problems with processing and integrating XML documents could be solved if we could associate machine understandable meaning with the tags. This meaning could be used to translate from one DTD to another, or reason about the consequences of a given set of facts. Knowledge representation, an important sub-field of artificial intelligence, can provide insights into these problems. A knowledge representation scheme describes how a program can model what it knows about the world. The goal of knowledge representation is to create schemes that allow information to be efficiently stored, modified, and reasoned with. Research in the field has spawned a number of knowledge representation languages, each with its own set of features and tradeoffs. These languages differ in the way that knowledge is acquired, the extent of the descriptions they provide, and the type of inferences that they sanction. An understanding of knowledge representation can provide key insights into the design of a language for the Semantic Web. In this section, we will consider a number of representations and formalisms that are particularly relevant.

2.2.1 Semantic Networks and Frame Systems

One of the oldest knowledge representation formalisms is semantic networks [79]. In a semantic net, each concept is represented by a node in a graph. Concepts that are semantically related are connected by arcs, which may or may not be labeled. In such a representation, meaning is implied by the way a concept is connected to other concepts.

Many semantic networks use special arcs to represent abstraction, although as Brachman [10] points out, the semantics of these links were often unclear. Now it is rather common to use two arcs for this purpose. An *is-a* arc indicates that one concept is subclass of another, while an *instance-of* arc indicates that a concept is an example of another concept. These arcs have correlations in basic set theory: *is-a* is like the subset relation and *instance-of* is like the element of relation.

The collection of *is-a* arcs specifies a partial order on classes; this order is often called a taxonomy or categorization hierarchy. The taxonomy can be used to generalize a concept to a more abstract class or to specialize a class to its more specific concepts. As demonstrated by the popularity of Yahoo and the Open Directory, taxonomies are clearly useful for aiding a user in locating relevant information on the Web. However, these directory taxonomies often deviate from the strict subset semantics followed by modern knowledge representation systems, making them less useful for automated reasoning.

In the 1970's, Minsky [73] introduced frame systems. In the terminology of such systems, a frame is a named data object that has a set of slots, where each slot represents a property or attribute of the object. Slots can have one or more values (called fillers), some of which may be pointers to other frames. Since each frame has a set of slots that represent its properties, frame systems are usually considered to be more structured than semantic networks. However, it has been shown that frame systems are isomorphic to semantic networks.

KRL [7] is an early knowledge representation language based on frame systems. The fundamental entities in KRL are units, which consist of a unique name, a category type, and one or more named slots, each of which can have its own description. Each class has a prototype individual which represents a typical member of the class. The language supports operations to add knowledge to a description, to determine if two descriptions are compatible and to find a referent that matches a given description. An interesting, but somewhat forgotten feature of KRL, was the ability to view an individual from different perspectives. For example, (the age from Person G0043) might be an integer, while (the age from Traveler G0043) might be an element from the set {Infant, Child, Adult}.

KL-ONE [13] continued the tradition of frame systems, while spawning the family of description logic systems, including Classic [12], LOOM [67], and FaCThorrocks:fact. Description logics focus on the definitions of terms in order to provide more precise semantics than semantic networks or earlier frame systems. Term definitions are formed by combining concepts and roles that can provide either necessary and sufficient conditions or just necessary conditions. A description is said to subsume another if it describes all of the instances that are described by the second description. An important feature of description logic systems is the ability to perform automatic classification, that is, automatically insert a given concept at the appropriate place in the taxonomy. The advantages of descriptions logics are they have well-founded semantics and the factors that affect their computational complexity are well understood, but it is unclear whether their inferential capabilities are the right ones for the Web.

Semantic nets and frame systems provide an intuitive basis from which to design a semantic web language. The SHOE language, which will be described in this thesis, makes extensive uses of class taxonomies. Although many knowledge representation systems cannot scale to the sizes needed for the Web, our applications have made extensive use of a scalable, high-performance system called Parka [27, 84], which will be described in detail in Section 5.2.7.

2.2.2 First-Order Logic

First-order logic (FOL), also known as predicate calculus or predicate logic, is a well-understood formalism for reasoning. Although the logic and knowledge representation communities are distinct, the expressivity of FOL nevertheless makes it a powerful knowledge representation language. From the perspective of FOL, the world consists of objects and the relations that hold between them.

A FOL language consists of logical and non-logical symbols. The logical symbols represent quantification, implication, conjunction and disjunction; while the non-logical symbols are constants, predicates, functions, and variables. Constant, variable and function symbols are used to build terms, which can be combined with predicates to construct formulas. A subset of the possible formulas that obeys certain rules of syntactic construction are called well-formed formulas.

The semantics of FOL are given by Tarski's model theory, referred to as model-theoretic or denotational semantics. In this treatment, an interpretation is used to relate the symbols of the language to the world. An interpretation consists of a set D of individuals called the domain of discourse, a function that maps constants symbols to D, a function that maps function symbols to functions on D, and a function that maps predicate symbols to relations on D. If a formula is true under some interpretation, then that interpretation is a model of the sentence. Likewise, if a set of formulas is true under some interpretation, then the interpretation is a model of the formulas. Given a set of formulas Γ, if some formula ϕ is necessarily true, then we say that Γ entails ϕ, written $\Gamma \models \phi$. Typically, there is a special interpretation, called the intended interpretation that accurately reflects the desired meaning for a set of sentences. A theory $\mathcal{T}$ is a set of sentences that are closed under logical implication. Thus, for all ϕ such that $\mathcal{T} \models \phi$, $\phi \in \mathcal{T}$. A good introduction to FOL can be found in Genesereth and Nilsson's textbook [39], while Lloyd [65] provides a more detailed treatment.

An inference procedure is an algorithm that can compute the sentences that are logically entailed by a knowledge base. FOL has a sound and complete inference procedure called resolution refutation. A sound procedure only generates entailed sentences, while a complete procedure can find a proof for any sentence that is entailed. However, refutation is intractable, making it a poor choice for reasoning with large knowledge bases.

The Knowledge Interchange Format (KIF) [41] is a standard language that can be used to exchange FOL sentences between different programs. However, KIF does not explicitly address the problems inherent in the decentralized definition of symbols needed on the Web. In order to define domain specific vocabularies, we need ontologies.

FOL is an extremely expressive representation, and can be used to describe semantic networks and frame systems. Due to this flexibility, when we describe a formal model of the Semantic Web in Chapter 3, we will use FOL as our basis.

2.2.3 Ontology

In order for information from different sources to be integrated, there needs to be a shared understanding of the relevant domain. Knowledge representation formalisms provide structures for organizing this knowledge, but provide no mechanisms for sharing it. Ontologies provide a common vocabulary to support the sharing and reuse of knowledge,

As discussed by Guarino and Giaretta [45], the meaning of the term ontology is often vague. It was first used to describe the philosophical study of the nature and organization of reality. In AI, the most cited definition is due to Tom Gruber [42]: "An ontology is an explicit specification of a conceptualization." In this definition, a conceptualization is an abstract view of the world, along the lines of Genesereth and Nilsson [39]. In particular, it is a tuple $\langle D, R \rangle$, where D is the domain of discourse and R is a set of relations on D. An ontology associates vocabulary terms with entities identified in the conceptualization and provides definitions to constrain the interpretations of these terms.

Guarino and Giaretta [45] argue that Genesereth and Nilsson's definition of conceptualization should not be used in defining ontology, because it implies that a conceptualization represents a single state of affairs (i.e., it is an extensional structure). However, an ontology should provide terms for representing all possible states of affairs with respect to a given domain. Therefore, they suggest that a conceptualization should be an intensional structure $\langle W, D, R \rangle$, where W is the set of possible worlds, D is the domain of discourse, and R is a set of intensional relations, where an n-ary intensional relation is a function from W to 2^{D^n} (the set of all possible n-ary relations on D). In a later paper, Guarino refines this model and provides the following definition for an ontology. [44]

> An ontology is a logical theory accounting for the intended meaning of a formal vocabulary, i.e., its ontological commitment to a particular conceptualization of the world. The intended models of a logical language using such a vocabulary are constrained by its ontological commitment. An ontology indirectly reflects this commitment (and the underlying conceptualization) by approximating these intended models.

Most researchers agree that an ontology must include a vocabulary and corresponding definitions, but there is no consensus on a more detailed characterization. Typically, the vocabulary includes terms for classes and relations, while the definitions of these terms may be informal text, or may be specified using a formal language like predicate logic. The advantage of formal definitions is that they allow a machine to perform much deeper reasoning; the disadvantage is that these definitions are much more difficult to construct.

Numerous ontologies have been constructed, with varying scopes, levels of detail, and viewpoints. Noy and Hafner [74] provide a good overview and comparison of some of these projects. One of the more prominent themes in ontology research is the construction of reusable components. The advantages of such components are clear: large ontologies can be quickly constructed by assembling and refining existing components, and integration of ontologies is easier when the ontologies share components.

One of the most common ways to achieve reusability is to allow the specification of an inclusion relation that states that one or more ontologies are included in the new theory. If these relationships are acyclic and treat all elements of the included ontology as if they were defined locally then an ontology can be said to extend its included ontologies. This is the case for most systems, however Ontolingua [30] has even more powerful features for reusability: inclusion relations that may contain cycles, the ability to restrict axioms, and polymorphic refinement.

Like an XML DTD or XML Schema, an ontology can provide a standard vocabulary for a problem domain. However, an ontology can also contain structures or axioms that define the semantics of the vocabulary terms. These semantics can be used to infer information based on background knowledge of the domain and to integrate data sources from different domains.

2.2.4 Context Logic

One of the problems with knowledge representation is that when we try to conceptualize some part of the world, we must make some simplifying assumptions about its structure. If we then try to combine knowledge bases (or logical theories), differences in their implicit, underlying assumptions may have unintended side-effects. Context logic [46, 68] proposes to solve this problem by explicitly placing each assertion in a context, where the context includes the assumptions necessary for the assertion to be true.

The assumptions of a knowledge base often determine the structure of its vocabulary. For example, an on-line retailer may choose to represent its catalog using *product(X, P)* where X is a product identifier and P is its price. However, this representation assumes a standard currency, perhaps U.S. dollars. If the retailer went international, they would need to change the representation to *product(X, P, C)*, where C identifies the currency. This representation still has an implied seller, and an intelligent shopping agent might want this information explicit, requiring instead a representation such as *sells(S, X, P, C)*, where S identifies the seller. We could continue to identify assumptions and expand the representation *ad infinitum* if so desired. Nevertheless, it may be more convenient for the retailer to provide its catalog in the *product(X, P)* form. But in order to do so, it must be possible to use the assumptions implicit in the form to convert to the other forms as necessary.

In context logic, contexts are first-class objects that can be used in propositions. Propositions of the form *ist(c,p)* are used to indicate that proposition p is true in context c. A particular individual i can be excluded from the scope of a context c by stating ¬*presentIn(c, i)*. The reification of context also makes it possible to combine information from many contexts. For example, one may wish to reuse parts of one context in another or make statements that are simultaneously true in a set of contexts. Statements that achieve these effects are called *lifting axioms*.

Another issue raised by context logic is that different contexts may contain mutually inconsistent assertions. Such situations should not lead to inconsistency of the entire knowledge base. Instead, context logic only requires a context to be locally consistent. This issue is of direct relevance to the Semantic Web, where knowledge is being provided by many users who may have inconsistent assumptions.

Context logic is implemented in Cyc [63, 64], an ongoing project with the ambitious goal of encoding the entirety of common sense. Contexts are represented by microtheories, which partition the knowledge base, and can extend one another using standard ontology inclusion principles. Since Cyc has an enormous ontology, microtheories are essential to its creation. They simplify the encoding of assertions by knowledge engineers, avoid the inevitable contradictions that arise from a large knowledge base, and help guide inferencing mechanisms by grouping relevant statements.

Ontologies and context logic are closely related. Each context is an ontology, and ontology inclusion could be one particular type of lifting axiom. An important aspect of context logic is that different contexts may be suitable for solving different problems. We will return to this point in Section 3.8.

2.3 Deductive Databases

One problem with many of the knowledge representation techniques discussed in Section 2.2 is that they do not scale well. However, if the size of the current web is an indication of the size the Semantic Web, then we know that any practical reasoning method must scale to enormous sizes. Deductive databases [72] extend traditional relational database techniques by allowing some of the relations to be computed from logical rules. Thus they combine the ability to perform inference with the ability to scale to large data sizes, both of which are required for the Semantic Web. Deductive databases address two deficiencies of logic programming languages such as Prolog [81]. First, Prolog's depth-first evaluation strategy requires careful construction of programs to avoid infinite loops. Second, efficient access to secondary storage is required to cope with a large volume of data.

A common logic-based data model is datalog [86, Chapter 3]. It is similar to Prolog in that it consists entirely of Horn clauses, but differs in that it does not allow function symbols and is a strictly declarative language.[1] In datalog, relations that are are physically stored in the database are called *extensional database* (EDB) relations and are identical to relations in the relational data model. The main difference between datalog and the relational model is that it also allows relations which are defined by logical rules, called *intensional database* (IDB) relations. Datalog also has a number of built-in predicates for standard arithmetic comparison. Any predicate that is not built-in is called ordinary.

Datalog relations are denoted by atomic formulas, which consist of a predicate symbol and a list of arguments. A argument can be either a constant or a variable. IDB relations are Horn clauses, which take the form h :- $b_1, b_2, \ldots, b_n$, where h and all b_i are atomic formulas. The left hand side h is called the head or consequent, and the right hand side is called the body, antecedents, or subgoals. The meaning of the rule is if the body is conjunctively true, then the head is also true. IDB relations may depend on each other recursively by containing each other in their bodies. Programs that contain such rules are called recursive.

In order keep relations finite, datalog defines the notions of limited variables and safety. A variable is *limited* if it appears in an ordinary predicate of the rule's body, appears in an '=' comparison with a constant, or appears in an '=' comparison with another limited variable. A rule is *safe* if all of its variables are limited.

An important branch of research in deductive databases deals with the optimization of queries. The standard methods for evaluating queries in logic are backward-chaining (or top-down) and forward-chaining (or bottom-up). In backward-chaining, the system uses the query as a goal and creates more goals by

[1]Prolog is not strictly declarative because the order of the rules determines how the system processes them.

expanding each head into its body. This approach ensures that only potentially relevant goals are explored but can result in infinite loops. Forward-chaining starts with the EDB and repeatedly uses the rules to infer more facts. As such, it avoids the problems of looping, but may infer many irrelevant facts. An important result from deductive databases is the magic sets technique [87], which rewrites rules so that a forward-chaining evaluation will only consider potentially relevant goals similar to those explored by backward-chaining.

Another prominent theme in deductive database research deals with allowing negation in the model. When negated literals are allowed, a program may not have a unique, minimal model, which is used to define the meaning of datalog programs. One form of negation that has intuitive semantics is *stratified negation*, in which negated subgoals are not used recursively.

Due to the focus of deductive databases on logic with large data sets, this work provides insight into practical implementations of the Semantic Web. The work described in this thesis makes extensive use of XSB [83], a deductive database that will be described in detail in Section 5.2.6.

2.4 Distributed Databases

If we can treat web pages as structured content, then it is possible to view the Web as a collection of autonomous databases. From this perspective, research in distributed databases is important. In this section, we will describe the themes of deductive database research and discuss how they might be relevant to the Semantic Web.

The degree of coupling between the components of a multidatabase system can be used to classify different architectures [9]. Global schema integration requires that the individual schemas of each database be merged, so that a single schema can be presented to users. Federated database systems (FDBSs) allow component databases to retain some degree of autonomy, and export portions of their schemas for use by the federation. In a tightly coupled FDBS, a single schema exists for the federation, and methods exist to translate between each export schema and the federation schema. In a loosely coupled FDBS, users create their own views from the export schemas. When maintaining the autonomy of the component databases is of chief importance, then the multidatabase language approach is used. No modifications are made to the participating databases, instead a special query language is is used to access and combine results from the different databases.

The problems addressed by distributed database research include integrating heterogeneous database management systems (DBMSs), concurrency control and transaction management of distributed databases, ensuring consistency of replicated data, query planning for accessing distributed data sources, and resolving the problems of semantic heterogeneity. We will discuss each of these issues in turn to determine its relevance to the Semantic Web. For a more detailed discussion of these issues in their original context, see the book by Elmagarmid, Rusinkiewicz, and Sheth [25].

The problem of heterogeneous DBMSs is integrating data contained in databases designed by different vendors that possibly use different data models (e.g., the relational model versus the object-oriented model). This is not a problem for the Semantic Web because the Web has a standard access protocol (HTTP) and ideally the Semantic Web will have a single language for exchanging knowledge (most likely based on XML).

Concurrency control and transaction management are core issues for all DBMSs. Concurrency control deals with allowing simultaneous access to a database while ensuring consistency of the database in the presence of potentially multiple writers. Transactions are used to ensure that a sequence of operations are treated as a unit. For example, when transferring money from one bank account to another, a transaction can be used to ensure that if the database failed in the middle of the transfer, money would not be lost or gained. These issues are very difficult for distributed databases due to the heterogeneity and autonomy of the component databases. Fortunately, they do not have much of an impact on the Semantic Web, since most operations are read-only. That is, typically the only updates to web pages are performed by their owners. Furthermore, each HTTP request is a separate transaction. However, if the Semantic Web is eventually used as a general infrastructure for e-commerce, then it may become necessary to treat a series of HTTP operations as a transaction.

When different copies of the same data are maintained in different locations, the data is said to be replicated. The problem with replication in distributed databases is that all updates must be made to each copy to ensure that the data is kept consistent. On the Web, it is certainly possible, even probable, that data will be replicated, but due to the autonomy of web sites, it is impossible to ensure that this data is consistent. In fact, since the Web is meant to represent the viewpoints of many, it is undesirable to treat it as a single, consistent database.

Of the problems faced by distributed databases, the most significant to the Semantic Web is semantic heterogeneity. Different database designers can model the world in many different ways, resulting in differences in naming, structure, and format. The autonomy of web sites will lead to similar problems with the Semantic Web.

Kashyap and Sheth [56] provide an overview of approaches to classifying and evaluating the semantic similarity of objects from different databases. An important component to many of these approaches is identifying a context. However, the contexts used are not necessarily the same as those in context logic. In the semantic proximity approach, the semantic similarity of two concepts is is defined by the kinds of contexts in which they are similar and the difference in the abstractions of their domains. However, the information provided is insufficient to automatically integrate two databases. In the context building approach, a set of interschema correspondence assertions (ISCAs) define a context. An ISCA states whether two terms are synonymous or polysemous, what difference in abstraction they represent and what kinds of structural differences exist. An ISCA can be used to perform some integration automatically, but lacks the expressive power to describe a full translation from one schema to another. The context interchange approach associates contextual metadata with attributes, and has conversion functions that can translate data from an export context to an import context by using the appropriate function to resolve the difference indicated in the metadata. In general, these approaches seem limited to resolving only some (if any) of the problems of semantic heterogeneity.

Other resource integration approaches use knowledge bases to perform schema integration. The Carnot [19] architecture uses the Cyc knowledge base as a global schema. A separate context is created for each component schema, and articulation axioms are used to state equivalence between objects in these schemas and the global one. Farquhar et al. [28] expand upon the use of contexts and articulation axioms, arguing that they can be used to achieve either a loosely-coupled federated database approach, or a global schema approach as needed. The advantage is that multidatabase-style access can be achieved quickly, and integration can be performed incrementally by adding more lifting axioms.

Closely related work in information integration focuses on building systems that can combine information from many different types of sources, including file systems, web pages, and legacy systems. Information integration systems typically have a mediator architecture [90], where mediators are components that serve as an interface between user applications and data sources. The mediator receives queries from the applications, determines which data sources contain the data necessary to answer the queries, and issues the appropriate queries to the sources. The data sources are encapsulated by wrappers [78, 82] which provide a uniform interface for the mediators. TSIMMIS [37], Ariadne [58], Infomaster [40], and Garlic [82] all follow this basic architecture. Most of these systems use some sort of logic language to describe data sources and translate between them. For example, TSIMMIS specifies mediators, wrappers, and queries using a variant of datalog, while Infomaster uses KIF to specify translations. With the appropriate wrappers, information integration systems can treat the Web as a database. However, the heterogeneity of the Web requires that a multitude of custom wrappers must be developed, and it is possible that important relationships cannot be extracted from the text based solely on the structure of the document. Semi-automatic generation of wrappers [60, 2] is a promising approach to overcoming the first problem, but is limited to data that has a recognizable structure. Another problem with existing mediator systems is that they require a single schema for specifying application queries. However, this means that if data sources begin providing new kinds of information, it will not become available in the system until the mediator and wrapper is updated.

2.5 Other Related Work

Querying the Web is such an important problem that a diverse body of research has been directed towards it. In the previous sections we tried to focus on the research that was most relevant to the topic of this thesis. In this section, we give a brief overview of different approaches to the problem.

Some projects focus on creating query languages for the Web [1, 59], but these approaches are limited to queries concerning the HTML structure of the document and the hypertext links. They also rely on index servers such as AltaVista or Lycos to search for words or phrases, and thus suffer from the limitations of keyword search.

Work on semistructured databases [70] is of great significance to querying and processing XML, but the semistructured model suffers the same interoperability problems as XML. Even techniques such as data guides will be of little use when integrating information developed by different communities in different contexts.

In order to avoid the overhead of annotating pages or writing wrappers, some researchers have proposed machine learning techniques. Craven et al. [20] have trained a system to classify web pages and extract relations from them in accordance with a simple ontology. However, this approach is constrained by the time-consuming task of developing a training set and has difficulty in classifying certain kinds of pages due to the lack of similarities between pages in the same class.

Chapter 3

A Logical Foundation for the Semantic Web

In this chapter, we will develop a framework for reasoning about the Semantic Web. We will start with a basic logic approach and gradually refine it to deal with the problems of representing knowledge on the Web.

3.1 An Initial Approach

A requirement for the Semantic Web is the ability to associate explicit meaning with the content of resources. We will do this by embedding a logical language in the resources and providing a denotational semantics for it. Many of the knowledge representation languages and structures discussed in Chapter 2, such as semantic networks, frame systems, and datalog, can all be formulated in first-order logic. For this reason, and because first-order logic is well-understood, we will use it as our basis. In order to use this framework with systems that cannot be described in first-order logic (e.g., probabilistic logics, temporal logics, higher-order logic), one must reformulate what follows to correspond to the desired logic.

First we must define our domain of discourse. The main objects of interest are internet resources and entities that are described by them. An internet resource is anything that provides information via the Internet, such as a web page, newsgroup, or e-mail message. We will use R to refer to the set of these resources. The domain of discourse, on the other hand, is the collection of things that are described or mentioned by internet resources, including potentially internet resources themselves. We will use D to refer to this set.

We will assume that we have a first-order language $\mathcal{L}$ with a set of non-logical symbols S. The predicate symbols of S are $S_P \subset S$, the variable symbols are $S_X \subset S$, and the constant symbols are $S_C \subset S$. For simplicity, we will not discuss function symbols, since an n-ary function symbol can be represented by a n+1-ary predicate. The well-formed formulas of $\mathcal{L}$ are defined in the usual recursive way. We will use W to refer to the infinite set of well-formed formulas that can be constructed in $\mathcal{L}$.

Let $K : R \rightarrow 2^W$ be a function that maps each resource into a set of well-formed formulas. We call K the *knowledge function* because it extracts the knowledge contained in a resource and provides an axiomatization for it.

We will define an interpretation $\mathcal{I}$ in the standard way. It consists of the domain of discourse D (as defined above), a function $\mathcal{I}_C : S_C \rightarrow D$ that maps constant symbols to elements of the domain, and a set of functions $\mathcal{I}_{P_n} : S_P \rightarrow D^n$ that map n-ary predicate symbols to sets of n-tuples formed from the domain. If a formula ϕ is true with respect to an interpretation $\mathcal{I}$, then we write $\mathcal{I} \models \phi$ and say that $\mathcal{I}$ satisfies ϕ or that $\mathcal{I}$ is a model of ϕ. Given a set of sentences Γ, if an interpretation $\mathcal{I}$ satisfies every $\phi \in \Gamma$ then we write $\mathcal{I} \models \Gamma$.

One way to consider the Semantic Web is to think of each resource as specifying an independent theory, that is, there is no interaction between the theories. In this approach, each resource must specify the complete theory that is needed to reason about it. For example, a genealogy web page that contains information on one's ancestors should include the following axioms that provide basic semantics for the $ancestorOf$ predicate:

$$parentOf(x, y) \rightarrow ancestorOf(x, y)$$

$$ancestorOf(x, y) \wedge ancestorOf(y, z) \rightarrow ancestorOf(x, z)$$

However, a disadvantage of the independent theory approach is that all other genealogy pages must replicate the axioms that define the basic genealogy predicates. Furthermore, if one resource contained the fact $ancestorOf(alice, bill)$ and another contained the fact $ancestorOf(bill, carol)$, then we would be unable to conclude $ancestorOf(alice, carol)$, because we cannot combine the theories.

In order to prevent the Semantic Web from becoming a billion unrelated islands, there needs to be a way to combine the information contained in the resources. We will state this as a fundamental principle of the Semantic Web:

Principle 3.1 *The Semantic Web must provide the ability to combine information from multiple resources.*

Given this proposition, let us consider an approach to combining the resources. We define a naive integrated theory NIT as the union of the well-formed formulas generated by the set of resources.

Definition 3.2 *Given a set of resources R, a naive integrated theory is:*

$$NIT(R) = \bigcup_{r \in R} K(r)$$

At first glance, this seems to be a sufficient approach. Since the formulas of all resources are combined, the axiomatization of any domain needs only to be expressed in a single resource, and it is possible to deduce things that require premises from distinct resources. However, upon closer inspection, problems with this approach begin to emerge.

Recall that the Web is a decentralized system and its resources are autonomous. As a result, different content providers are free to assign their own meanings to each nonlogical symbol, thus it is likely that multiple meanings will be assigned to many symbols. Different axiomatizations for the same symbols may result from the polysemy of certain words, poor modeling, or even malicious attempts to break the logic.

To resolve the problem of accidental name conflicts, we will assume that the constants S_C of $\mathcal{L}$ are URIs. Since URIs provide hierarchical namespaces (as described in in Section 2.1.2), they can be used to guarantee that constants created by different parties will be distinct.

Other problems are more complex. As pointed out by Guha [46, Section 2.6], a theory usually has an implied background theory that consists of its assumptions. To combine a set of theories accurately, we need to make some of these assumptions explicit. For example, if one theory about stock prices assumed that the day was yesterday and another assumed that the day was today, an integrated theory should attach the date to each assertion. Guha calls this process *relative decontextualization*. Relative decontextualization may also involve mapping synonymous terms, or in the more complex case, different representational structures that have the same meaning. Note that in order to perform relative decontextualization, one must first know the contexts associated with the two theories. We will distinguish between simple combination, in which no relative decontextualization is performed, and integration, in which it is performed.

Principle 3.3 *Semantic web resources can only be integrated after they have undergone relative decontextualization.*

One way to avoid the need for relative decontextualization is to create a standardized vocabulary with official definitions for each symbol, However, to handle all expressions that might appear on the Web, the vocabulary would have to be enormous, making it nearly impossible to standardize, comprehend, and later change as necessary.

3.2 An Ontology-Based Approach

Recall from Section 2.2.3 that an ontology provides a common vocabulary to support the sharing and reuse of knowledge. When two parties agree to use the same ontology, they agree on the meanings for all terms from that ontology and their information can be combined easily. Unfortunately, there is no widely accepted formal definition of an ontology. In this section and the next two, we will formally define ontologies that are applicable to the Semantic Web.

3.2.1 Ontology Definitions

Let us think of an ontology as simply a set of symbols and a set of formal definitions, along the lines of Farquhar, Fikes, and Rice [30]. We will assume that the formal definitions are written in the language $\mathcal{L}$. We can now define an ontology:

Definition 3.4 *Given a logical language $\mathcal{L}$, an ontology is a tuple $\langle V, A \rangle$, where the vocabulary $V \subset S_P$ is some subset of the predicate symbols of $\mathcal{L}$ and the axioms $A \subset W$ are a subset of the well-formed formulas of $\mathcal{L}$.*

As a result of this definition, an ontology defines a logical language that is a subset of the language $\mathcal{L}$, and defines a core set of axioms for this language. Since the ontology defines a language, we we can talk about well-formed formulas with respect to an ontology.

Definition 3.5 *A formula ϕ is well-formed with respect to an ontology $O = \langle V, A \rangle$, iff ϕ is a well-formed formula of a language $\mathcal{L}'$, where the constant symbols are S_C and the variable symbols are S_X, but the predicate symbols are V.*

We can also define what it means for an ontology to be well-formed.

Definition 3.6 *An ontology $O = \langle V, A \rangle$ is well-formed if every formula in A is well-formed with respect to O.*

We will now provide meaning for ontologies by defining interpretations and models for them. First, we define a pre-interpretation that maps each constant symbol of the language to the domain of discourse.

Definition 3.7 *A pre-interpretation of $\mathcal{L}$ is a structure that consists of a domain D and a function that maps every constant symbol in S_C to a member of D.*

Every $\mathcal{L}$-ontology uses the same pre-interpretation. Since the symbols from S_C are URIs, their intended interpretation is fixed by the URI scheme or by their owners. For this reason, it is assumed that these interpretations are universal. An interpretation of an ontology consists of the pre-interpretation and a mapping of the predicate symbols of $\mathcal{L}$ to relations on the domain.

Definition 3.8 *An interpretation of an ontology consists of a pre-interpretation and a function that maps every n-ary predicate symbol in S_P to an n-ary relation on D.*

We can now define a model of an ontology.

Definition 3.9 *A model of an ontology O=*$\langle V, A\rangle$ *is an interpretation that satisfies every axiom in A.*

Thus an ontology attempts to describe a set of possibilities by using axioms to limit its models. Some subset of these models are those intended by the ontology, and are called the intended models of the ontology. Note that unlike a first-order logic theory, an ontology can have many intended models because it can be used to describe many different states of affairs.

Note that we chose to have the interpretation of an ontology assign relations to every predicate symbol in the language $\mathcal{L}$, not just those in the ontology. This makes it possible to compare the models of different ontologies that may have separate vocabularies. Since we are treating ontologies as disjoint, this is not significant now. However, it will become important when we begin to discuss ontologies that can extend other ontologies and reuse their vocabulary. Also note that the intended interpretations of an ontology will limit the relations that directly or indirectly correspond to predicates in its vocabulary, while allowing any of the possibilities for predicate symbols in other domains.

3.2.2 Resource Definitions

Now we need to associate an ontology with each resource. If we let $\mathcal{O}$ be the set of ontologies, then we can create a function $C : R \to \mathcal{O}$, which maps resources to ontologies. We call this the *commitment function* because it returns the ontology that a particular resource commits to. When a resource commits to an ontology, it agrees to the meanings ascribed to the symbols by that ontology. Because C is a function, a resource can only commit to a single ontology, but in Section 3.3.3 we will show how a single ontology can combine multiple ontologies, thus overcoming this limitation.

The vocabulary that a resource may use is limited by the ontology to which it commits.

Definition 3.10 *A resource r is well-formed if $C(r) = O$ and $K(r)$ is well-formed with respect to O.*

That is, a resource is well-formed if the theory given by the knowledge function is well-formed with respect to the ontology given by the commitment function.

We now wish to define the semantics of a resource. When a resource commits to an ontology, it has agreed to the terminology and definitions of the ontology. Thus every interpretation of an ontology is an interpretation of the resources that commit to it, and an interpretation that also satisfies the formulas of a resource is a model of that resource.

Definition 3.11 *A model of a resource r, where $C(r) = O$, is a model of O that also satisfies every formula in $K(r)$.*

3.2.3 Simple Ontology Perspectives

Using the definitions above, we could once again create a separate theory for each resource, knowing that the ontologies provide reusable sets of axioms that do not need to be repeated for each resource in the same domain. However, this approach would still prevent us from combining information from different resources, and thus be in conflict with Principle 3.1. Instead, we will consider a way to create larger theories that combine resources which share ontologies. We will attempt to divide the Semantic Web into sets of resources that share a context, and thus can be combined without relative decontextualization. We will call these divisions perspectives, because they provide different views of the Semantic Web.

We need some guidelines for determining how to construct the perspectives. Each perspective will be based on an ontology, hereafter called the basis ontology or base of the perspective. By providing a set of terms and a standard set of axioms, an ontology provides a shared context. Thus, resources that commit to the same ontology have implicitly agreed to share a context. To preserve the semantics intended by the author of each ontology and resource, we will require that the models of each perspective be a subset of the models of its basis ontology and of each resource included in the perspective. Thus, the perspective must contain the axioms of the ontology and the formulas of each resource that commits to it.

Definition 3.12 *Given a set of ontologies $\mathcal{O} = \{O_1, O_2, \ldots, O_n\}$ where $O_i = \langle V_i, A_i \rangle$, a simple ontology perspective based on ontology O_i is:*

$$SOP_i(R) = A_i \cup \bigcup_{\{r \in R | C(r) = O_i\}} K(r)$$

With this approach, we have a separate logical theory for each ontology. Each of these theories includes the axioms of the ontology that serves as its basis and the theories of each resource that commits to that ontology. Since each resource in the perspective agrees to the meanings ascribed to the symbols by the perspective's ontology, there will be no name conflicts between different resources in the perspective. Additionally, since only one ontology is used in each perspective, there is no possibility of axioms from another ontology having unintended side effects.

Although the simple ontology perspective approach solves many problems, it greatly restricts interoperability of resources. The only resources that can be integrated are those that commit to the same ontology. Obviously, this is too restrictive; some content providers may find that an existing ontology would be suitable if only a few minor additions were made. It would be unfortunate if a new, incompatible ontology had to be created for this purpose. Furthermore, if information concerning a domain is provided independently from different sources, then it is likely that the sources will want to use the terminology that is convenient for them. These needs can be summed up in a single principle.

Principle 3.13 *A semantic web ontology should be able to extend other ontologies with new terms and definitions.*

This principle requires that ontologies be able to reference other ontologies, and to provide axioms that relate new terms to the terms in these other ontologies.

3.3 Ontology Extension

Two prominent themes in ontology research are reusability and composability. It is recognized that ontology construction is a difficult and time-consuming task, so the ability to create standard modules for specific purposes is appealing. In theory, if standard ontologies existed for common tasks, much of the ontology design process could occur by assembling a number of existing modules, and simply modeling the unique aspects of the domain as needed. A useful consequence of ontology reuse is that all ontologies that reuse a given module will use the same vocabulary and axioms to model similar concepts. In the previous section, we described how different resources can reuse ontologies, but now we will consider ontologies that reuse other ontologies.

In most existing ontology work, reuse is handled by providing a mechanism that allows an ontology to extend another ontology (see Section 2.2.3). Essentially, when an ontology extends another, it includes the axioms and vocabulary of that ontology and any ontology extended by the second ontology. Using this framework, it is possible to design taxonomies of ontologies, with high-level generic ontologies at the top, and more specific ontologies at the bottom. Thus it is possible to have a top-level ontology that defines common concepts such as *Person* and *Organization*, which is extended by industry specific ontologies, that are in turn extended by corporation specific ontologies, and so on.

Unlike work in schema integration, ontology extension integrates ontologies at schema (ontology) design time. When a new ontology is created, its relationships to other ontologies are specified. This process greatly reduces the semantic heterogeneity of the ontologies, while accommodating differences where necessary. When an existing term is needed, it is simply borrowed from another ontology; when the terms from other ontologies are unsuitable, a new term can be created and axioms can be used to describe its relationship to existing terms. Section 3.5 discusses the practical problems of such an approach.

3.3.1 Ontology Extension Definitions

We can formally define ontology extension as follows:

Definition 3.14 *Given ontologies O_1 and O_2, O_1 is said to extend O_2 iff all models of O_1 are also models of O_2.*

Note that this definition depends on the ability to compare the models of two different ontologies. Recall from Section 3.2.1 that the predicate interpretation function of an ontology is defined for all predicate symbols of $\mathcal{L}$, not just those in the ontology's vocabulary. Thus since every interpretation of every ontology has a set of tuples associated with each predicate symbols, it is possible to compare the tuples for each predicate symbol and determine whether one interpretation is a subset of another.

Now let us add the concept of ontology extension to our formalism. We will refine our definition of an ontology to include the set of ontologies extended by it. Definition 3.4 can be thought of as a special case of this definition, where $O = \langle V, A, \emptyset \rangle$.

Definition 3.15 *Given a logic $\mathcal{L}$, an ontology is a three-tuple $\langle V, A, E \rangle$, where the vocabulary $V \subset S_P$ is some subset of the predicate symbols, the axioms $A \subset W$ are a subset of the well-formed formulas, and $E \subset \mathcal{O}$ is the set of ontologies extended by O.*

This new ontology definition requires us to reconsider the definitions from Section 3.2. Many of the definitions are unchanged, but well-formedness with respect to an ontology, the well-formedness of ontologies, and the models of an ontology need to be redefined. However, before we can discuss the new definitions, we need to consider the uniqueness of the vocabulary symbols used by ontologies and define the concept of ancestor ontologies.

When different ontologies are used, they may assign different meanings to the same symbol. In the previous section, we ignored this fact because each ontology was used to form a separate theory. However, when ontologies include other ontologies, reasoners that assume that a symbol means the same thing when used in different contexts risk incorrect inferences. Therefore we will assume that unless otherwise stated, identical symbols in different ontologies represent distinct concepts, as advocated by Wiederhold [92]. In our framework, we will achieve this by prefixing each predicate symbol with its source ontology and a colon, for example *ont*:*symbol*. When we refer to symbols, we will use either qualified names or or unqualified names, where a qualified name includes the prefix, while an unqualified name does not. To prevent ambiguity, all unqualified names have exactly one corresponding qualified name, which is the name formed by adding the prefix of the ontology in which the name appears, or in the case of resources, the ontology committed to by the resource. The function $Q : name \rightarrow qname$ performs this mapping, where $qname \subset name \subset S_P$ and $\forall x, x \in qname \leftrightarrow Q(x) = x$. For convenience, we will write the set of names resulting from applying Q to each member of a set N as $Q(N)$.

An ancestor of an ontology is an ontology extended either directly or indirectly by it. If O_2 is an ancestor of O_1, we write $O_2 \in anc(O_1)$. In this case, we may also say that O_1 is a descendant of O_2. The formal definition of an ancestor is:

Definition 3.16 *Given ontologies* $O_1 = \langle V_1, A_1, E_1 \rangle$ *and* $O_2 = \langle V_2, A_2, E_2 \rangle$, $O_2 \in anc(O_1)$ *iff* $O_2 \in E_1$ *or there exists an* $O_i = \langle V_i, A_i, E_i \rangle$ *such that* $O_i \in E_1$ *and* $O_2 \in anc(O_i)$.

An ontology should have access to all symbols defined in its ancestors, and likewise a formula of that ontology should still be well-formed if it uses symbols from the ancestor ontologies. Thus, we need to redefine what it means to be well-formed with respect to an ontology. First, we must identify the vocabulary accessible to an ontology. This is the union of its own vocabulary and that of all of its ancestors. The vocabulary of an ontology is a set of unqualified names, but the extended vocabulary can be a mixture of qualified and unqualified names. This is because we must use the qualified names of the ancestor ontologies to guarantee that there are no name conflicts.

Definition 3.17 *The extended vocabulary* V_i^* *of an ontology* $O_i = \langle V_i, A_i, E_i \rangle$ *is* $V_i \cup \bigcup_{\{j | O_j \in anc(O)\}} Q(V_j)$.

Now we modify Definition 3.5 to say that a formula is well-formed with respect to an ontology if it is well-formed with respect to a language where the predicate symbols are given by the extended vocabulary of the ontology.

Definition 3.18 *A formula ϕ is well-formed with respect to an ontology $O = \langle V, A, E \rangle$ iff ϕ is a well-formed formula of a language $\mathcal{L}'$, where the constant symbols are S_C and the variable symbols are S_X, but the predicate symbols are V^*.*

We also need to modify the definition of a well-formed ontology. In addition to using the new definition of well-formedness with respect to an ontology, we must consider aspects of the extended ontologies. Besides requiring that they be well-formed, we also must prevent cycles of ontology extension.

Definition 3.19 *An ontology $O = \langle V, A, E \rangle$ is well-formed iff A is well-formed with respect to O, all ancestors of O are well-formed, and O is not an ancestor of O.*

Finally, let us redefine a model of an ontology. In particular, all models of an ontology should also be models of every ontology extended by it.

Definition 3.20 *Given an ontology $O = \langle V, A, E \rangle$, if $E = \emptyset$ then a model of O is an interpretation that satisfies every formula in A, otherwise a model of O is a model of every ontology in E that also satisfies every formula in A.*

3.3.2 Example of Ontology Extension

In Figure 3.1, we demonstrate how ontology extension can be used to relate the vocabularies of different domains, thus promoting interoperability. When two ontologies need to refer to a common concept, they should both extend an ontology in which that concept is defined. In this way, consistent definitions can be assigned to each concept, while still allowing communities to customize ontologies to include definitions and rules of their own for specialized areas of knowledge.

$$
\begin{aligned}
O_G = \quad & \langle \{Thing, Person, Object\}, \\
& \{Person(x) \rightarrow Thing(x), \\
& Object(x) \rightarrow Thing(x)\}, \\
& \emptyset \rangle \\
O_U = \quad & \langle \{Chair\}, \\
& \{Chair(x) \rightarrow O_G : Person(x)\}, \\
& \{O_G\} \rangle \\
O_F = \quad & \langle \{Chair\}, \\
& \{Chair(x) \rightarrow O_G : Object(x)\}, \\
& \{O_G\} \rangle \\
O_{U2} = \quad & \langle \{DeptHead\}, \\
& \{DeptHead(x) \leftrightarrow O_U : Chair(x)\}, \\
& \{O_U\} \rangle \\
O_{F2} = \quad & \langle \{Seat\}, \\
& \{Seat(x) \leftrightarrow O_F : Chair(x)\}, \\
& \{O_F\} \rangle
\end{aligned}
$$

Figure 3.1: Example of ontology extension.

The problems of synonymy and polysemy can be handled by the extension mechanism and use of axioms. An axiom of the form $P_1(x_1, \ldots, x_n) \leftrightarrow P_2(x_1, \ldots, x_n)$ can be used to state that two predicates are equivalent. With this idiom, ontologies can create aliases for terms, so that domain-specific vocabularies can be used. For example, in Figure 3.1, the term *DeptHead* in O_{U2} means the same thing as *Chair* in O_U due to an axiom in O_{U2}. Although this solves the problem of synonymy of terms, the same terms can still be used with different meanings in different ontologies. This is not undesirable, a term should not be restricted for use in one domain simply because it was first used in a particular ontology. As shown in the figure, different ontologies may also use the same term to define a different concept. Here, the term *Chair* means different things in O_U and O_F because different axioms are used to define it.

Figure 3.1 is easier to understand when shown graphically as in Figure 3.2. In this figure, we have assigned meaningful names to each ontology and used arcs to indicate

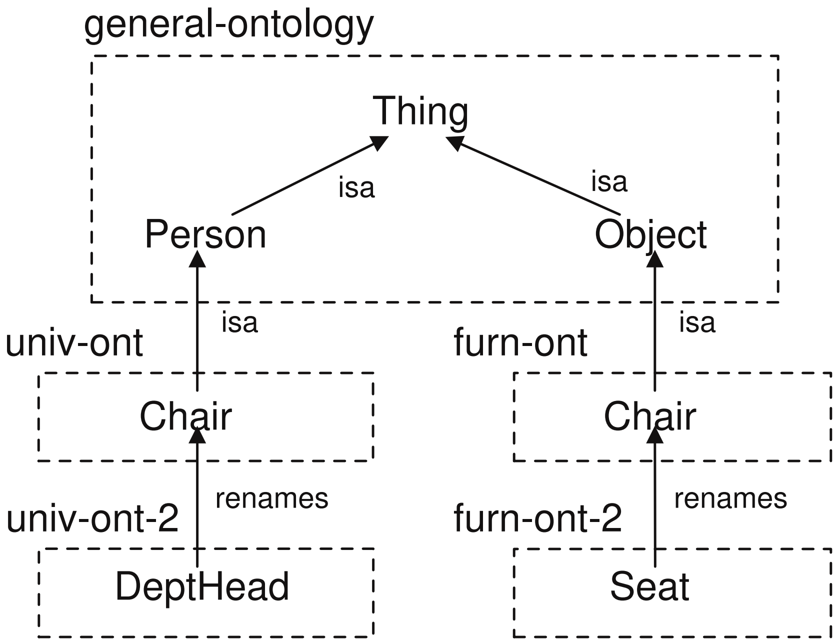

Figure 3.2: Graphical depiction of Figure 3.1.

two common types of axioms: *renames* is used for axioms that state two predicates are equivalent and *isa* is used for axioms of the form $C(x) \rightarrow P(x)$, to go along with the intuition that this means that all members of a class C are also members of a class P. We will say more on the usage of idioms in a semantic web language in Chapter 4.

3.3.3 Extended Ontology Perspectives

If we include ontology extension in our semantic web language, then how does that affect interoperability? Often, traditional ontology work has assumed that reuse was a mechanism to ease the construction of a single unified ontology. It had not considered that a number of related ontologies might be used to structure different data sets. For example, if two resources commit to different ontologies, where one ontology is an ancestor of the other, then it should be possible to integrate data from these sources.

However, since different ontologies can be provided by different sources, it is important that new ontologies do not automatically require a reinterpretation of existing data. Thus an extending ontology should provide the ability to reason about existing resources in new ways, but should not supersede the ontology that it extends. Otherwise, accidental or malicious ontologies could have serious side effects on existing portions of the Web. This point is important enough to deserve a principle.

Principle 3.21 *Each ontology should provide a different perspective on a set of resources, and no ontology should change the perspective of another.*

Given this principle and our new definition of an ontology, how can we define a perspective that maximizes integration? We will assume that an ontology which includes another does not attempt to change the intended meaning of the ontology and will contain any axioms necessary for decontextualization with respect to it. Thus, we can refine our ontology perspectives from Section 3.2 to include resources that commit to any ontologies that are ancestors of the perspective's basis ontology. We wish for our perspectives to be the intersection of the models of the ontology and all included resources. In the case of an ontology, its models are determined by its axioms and those of its ancestors, while the models of a resource are determined by its knowledge function and the ontology to which it commits. Thus, a new kind of perspective an be defined as follows:

Definition 3.22 *Given a set of ontologies $\mathcal{O} = \{O_1, O_2, \ldots, O_n\}$ where $O_i = \langle V_i, A_i, E_i \rangle$, then an extended ontology perspective based on ontology O_i is:*

$$EOP_i(R) = A_i \cup \bigcup_{\{j|O_j \in anc(O_i)\}} A_j \cup \bigcup_{\{r \in R|C(r)=O_i \vee C(r) \in anc(O_i)\}} K(r)$$

With extended ontology perspectives, there is a separate theory for each ontology, but some theories may have overlapping axioms and ground atoms, depending on how the basis ontologies are related. A perspective contains the axioms of its basis ontology, the axioms of all of its ancestor ontologies, and the formulas from all resources that commit to the defining ontology or one of its ancestors. Thus, two perspectives that are based on ontologies with a common ancestor will have that ancestor's axioms in common, and will have the formulas of all resources that commit to that ancestor in common as well.

A desirable property of these perspectives is that a perspective based on an ontology entails all of the sentences that are entailed by any perspectives based on ancestors of the ontologies. This ensures that any conclusions sanctioned by a resource are valid conclusions in any perspective that includes the resource. We will show that extended ontology perspectives satisfy this property.

Theorem 3.23 *Given two ontologies O_1 and O_2 such that $O_2 \in anc(O_1)$, if $EOP_{O_2}(R) \models \phi$, then $EOP_{O_1}(R) \models \phi$.*

Proof Let $O_1 = \langle V_1, A_1, E_1 \rangle$ and $O_2 = \langle V_2, A_2, E_2 \rangle$. We will prove the theorem by showing that $EOP_{O_1}(R)$ is a superset of each of the parts of $EOP_{O_2}(R)$. First, because $O_2 \in anc(O_1)$, $A_2 \in EOP_{O_1}(R)$. Second, due to Definition 3.16, $anc(O_1) \supset anc(O_2)$, thus the A_i of all ontologies $O_i \in anc(O_2)$ are also $\in EOP_{O_1}(R)$. Finally, since $O_2 \in anc(O_1)$ and $anc(O_1) \supset anc(O_2)$, all $K(r)$ such that $C(r) = O_2$ or $C(r) \in anc(O_2)$ are in $EOP_{O_1}(R)$. Therefore, $EOP_{O_1}(R) \supseteq EOP_{O_2}(R)$. Since FOL is monotonic, if $EOP_{O_2}(R) \models \phi$, then $EOP_{O_1}(R) \models \phi$.

As it turns out, simple ontology perspectives are a special case of extended ontology perspectives. When no relevant ontologies extend any other ontologies, the two functions are equivalent.

Theorem 3.24 *Given a set of ontologies $\mathcal{O} = \{O_1, O_2, \ldots, O_n\}$, where $\forall O_i \in \mathcal{O}, O_i = \langle V_i, A_i, \emptyset \rangle$, then $EOP_i(R) = SOP_i(R)$.*

Proof If we substitute $\emptyset$ for E_i in Definition 3.22, then $anc(O_i) = \emptyset$ and the set of j such that $O_j \in anc(O_i)$ is empty. Therefore, the corresponding union is $\emptyset$. Additionally, since C is a total function, the set of r such that $C(r) \in anc(O_i)$ is empty, which reduces the union condition to $\{r \in R | C(r) = O_i\}$. Thus the definition reduces to:

$$
\begin{aligned}
EOP_i(R) &= A_i \cup \bigcup_{\{j | O_j \in anc(O_i)\}} A_j \cup \bigcup_{\{r \in R | C(r) = O_i \vee C(r) \in anc(O_i)\}} K(r) \\
&= A_i \cup \emptyset \cup \bigcup_{\{r \in R | C(r) = O_i\}} K(r)
\end{aligned}
$$

$$
\begin{aligned}
&= A_i \cup \bigcup_{\{r \in R | C(r) = O_i\}} K(r) \\
&= OPT_i(R)
\end{aligned}
$$

Extension is useful in overcoming one of the limitations imposed by the commitment function. This function only allows each resource to commit to a single ontology, however with extension, a virtual ontology can be created that represents multiple ontologies committed to by a single resource. For example, given two ontologies $O_1 = \langle V_1, A_1, E_1 \rangle$ and $O_2 = \langle V_2, A_2, E_2 \rangle$, the ontology $O_{union} = \langle \emptyset, \emptyset, \{O_1, O_2\} \rangle$ is equivalent to their union. A resource that needed to commit to O_1 and O_2 could instead commit to O_{union}.

3.4 Ontology Evolution

The Web is a dynamic place, where anyone can instantaneously publish and update information. It is important that this ability is not lost when we provide more structure for the information. People must be able to publish semantic web ontologies as easily as other documents, and they must be allowed to revise these ontologies as well. While good design may prevent many ontological errors, some errors will not be realized until the ontology is put to use. Furthermore, pressing information needs may limit the time that can be applied to design particular ontologies, resulting in the need to improve the ontologies later. More philosophical arguments concerning the need for ontology revision are made by Foo [34].

Most ontology systems do not manage the problem of ontology change. Often this is because these systems are prototypes used for research purposes, and thus any dependencies are insignificant. For centralized systems, a change to the ontology can be synchronized with the corresponding changes to any dependent information, making change management unnecessary.

The developers of Ontolingua, a language used in the distributed development of ontologies, made the decision to ignore prior versions of an ontology. Farquhar, Fikes, and Rice [30] state that the inclusion feature in Ontolingua does not result in a "cut and paste" of the contents because "this interpretation would result in unfortunate version dependencies." However, this ignores the problem that the included ontology could change in a way that would make all of its including ontologies invalid.

One area where the problem of ontology change has been examined are medical terminology systems. Medical terminologies often consist of hierarchies of concepts, and sometimes include synonyms and properties. A number of different systems are used for different purposes, and the terminologies are frequently merged or mapped to each other, so that information from different systems can be combined. However, due to a number of factors, such as new medical knowledge and corrections of errors, the individual terminologies will continue to evolve. Since these terminologies are used in real systems, management of ontology change is a critical issue. Oliver et al. [77] discuss the kinds of changes that occur in medical ontologies and propose the CONCORDIA concept model to cope with these changes. The main aspects of CONCORDIA are that all concepts have a permanent unique identifier, concepts are given a *retired* status instead of being physically deleted, and special links are maintained to track the retired parents and children of each concept. However, this approach is insufficient for managing change on the Semantic Web. In the next sections, we will discuss the kinds of changes that might occur, and present a revised ontology definition that can describe these changes.

3.4.1 Ontology Evolution Examples

When we decide to change an ontology, then we must consider that in a distributed ontology framework such as the one needed by the Semantic Web, there will often be dependencies on it. We will illustrate the issues with examples of ontology change within our framework. In Figure 3.3, we demonstrate what happens when a new term is added to the ontology. In the example, O_U, r_1, and r_2 represent a simple university ontology and two resources that commit to it. Recall that an ontology is three-tuple $\langle V, A, E \rangle$ where V is its vocabulary, A is its set of axioms, and E is the set of ontologies extended by it. Also recall that K is the knowledge function that maps resources to formulas. Thus, O_U consists of a single term *Faculty*, while r_1 and r_2 are resources that use the *Faculty* predicate. At some later point in time, O'_U, r'_1, r'_2, and r'_3 represent the state of relevant web objects. Here, the ontology O'_U represents a new version O_U which includes terms that represent subclasses of *Faculty*. When an ontology designer adds terms in this way, it is likely that he will add axioms, such as $Professor(x) \rightarrow Faculty(x)$ to help define the terms. Note that r'_1 and r'_2 are r_1 and r_2, respectively at the later point in time. Because $K(r'_1) = K(r_1)$ and $K(r'_2) = K(r_2)$, these resources have not changed. Since the vocabulary V' of O'_U is a superset of V, r_1 and r_2 are still well-formed with respect O'_U. Once O_U has been revised to O'_U, we can create resources that use the new terms in O'_U; r'_3 is an example of such a resource that contains an assertion about *drjones*. All of the resources can be integrated with an extended ontology perspective. For example, if $R = \{r'_1, r'_2, r'_3\}$, then in $EOP_{O'_U}(R)$, *Faculty(drdoe)*, *Faculty(drsmith)*, and *Faculty(drjones)* are all true. Since we are assuming a monotonic logic, if we add terms and axioms to an ontology, then we know that the logical consequences of a perspective based on it will either be unchanged or increased.

$$
\begin{array}{ll}
O_U = & \langle\{Faculty\}, \\
& \emptyset, \\
& \emptyset\rangle \\
K(r_1) = & \{Faculty(drdoe)\} \\
K(r_2) = & \{Faculty(drsmith)\}
\end{array}
$$

$$
\begin{array}{ll}
O'_U = & \langle\{Faculty, AssistProf, AssocProf, Professor\}, \\
& \{AssistProf(x) \rightarrow Faculty(x), \\
& AssocProf(x) \rightarrow Faculty(x), \\
& Professor(x) \rightarrow Faculty(x)\} \\
& \emptyset\rangle \\
K(r'_1) = & \{Faculty(drdoe)\} \\
K(r'_2) = & \{Faculty(drsmith)\} \\
K(r'_3) = & \{AssocProf(drjones)\}
\end{array}
$$

Figure 3.3: Adding terms to an ontology.

However, if a term is deleted from the ontology then existing resources may become ill-formed. An example of this is presented in Figure 3.4. Here, we have a simple music store ontology that defines the classes *Cd* and *Tape*. Resource r_1 makes an assertion about an instance that is a *Cd*, while resource r_2 makes an assertion about an instance that is a *Tape*. Assume that at some point in the future, tapes become obsolete, and the decision is made to remove *Tape* from the ontology. If resource r_2 is not changed, then it becomes ill-formed because the ontology it commits to no longer includes the predicate used in its assertion. Since the resource may be not be owned by the ontology designer (for example, if it is a specific record store that is reusing the ontology), it is impossible to ensure that it will be updated when the ontology changes.

This leads us to another principle:

Principle 3.25 *The revision of an ontology should not change the well-formedness of resources that commit to an earlier version of the ontology.*

In practice, strict deletion will probably occur rarely. It is more likely that a term will be removed because it can be merged with another term, or a different name is

$$
\begin{aligned}
O_M = \quad & \langle \{Cd, Tape\}, \\
& \emptyset, \\
& \emptyset \rangle \\
K(r_1) = \quad & \{Cd(whiteAlbum)\} \\
K(r_2) = \quad & \{Tape(darkSide)\} \\
\\
O'_M = \quad & \langle \{Cd\}, \\
& \emptyset, \\
& \emptyset \rangle \\
K(r'_1) = \quad & \{Cd(whiteAlbum)\} \\
K(r'_2) = \quad & \{Tape(darkSide)\}
\end{aligned}
$$

Figure 3.4: Deleting a term from an ontology.

$$
\begin{aligned}
O_U = \quad & \langle \{Class\}, \\
& \emptyset, \\
& \emptyset \rangle \\
K(r_1) = \quad & \{Class(ai)\} \\
K(r_2) = \quad & \{Class(databases)\} \\
\\
O'_U = \quad & \langle \{Class, Course\}, \\
& \emptyset, \\
& \emptyset \rangle \\
K(r'_1) = \quad & \{Class(ai)\} \\
K(r'_2) = \quad & \{Class(databases)\} \\
K(r'_3) = \quad & \{Class(algFall2001)\} \\
K(r'_4) = \quad & \{Course(algorithms)\}
\end{aligned}
$$

Figure 3.5: Changing the meaning of a term from an ontology.

preferred. If the meaning of the term is changed, then a significant problem can arise. For example, consider Figure 3.5, where *Class* used to mean "a subject matter of instruction," but was changed to mean "a particular offering of a course," so that *Course* could be used for the old meaning. In this case, old resources such as r'_1 and r'_2 that used the term would be using it incorrectly in the context of the new ontology. However, since they would still be well-formed, there is no way to automatically detect the problem. As a result, false conclusions may be drawn from the information.

One possible solution to the ontology evolution problem is to require that revisions of ontologies be distinct ontologies in themselves. Then each resource can commit to a particular version of an ontology. For example, if in Figure 3.4, $C(r'_1) = O_U$ and $C(r'_2) = O_U$, then both resources commit to the version of the ontology that still has the term *Tape*. Since the ontology committed to by the resources does not physically change, they cannot become ill-formed unless the ontology changes.

Although treating each ontology version as a separate ontology solves the problems with deleting terms, it creates problems for integrating data from resources that commit to different versions of the ontology. Consider the example in Figure 3.3. Here, $C(r'_1) = O_U$, $C(r'_2) = O_U$ and $C(r'_3) = O'_U$. Because O_U and O'_U are different ontologies, and neither extends the other, then any resources that commit to them would be partitioned in separate theories. That is, the vocabularies are treated as distinct even though in fact they are just different formalizations of the same concept. We will formulate the need to integrate resources that commit to different versions of an ontology as a principle.

Principle 3.26 *Resources that commit to a revised ontology can be integrated with resources that commit to compatible prior versions of the ontology.*

In this principle, we need to define what is meant by "compatible." In cases such as the one examined in Figure 3.3, the newer version of an ontology is usually a better formalization of the domain than a previous version, (i.e., it is a closer approximation of the intended models). Thus, it would be useful if we could use the new perspective to reason about resources that committed to the older version. However, to do this, we need some way to indicate when the ontology revision is simply a refinement of the original ontology. In the next section, we augment our definition of ontology for this purpose.

3.4.2 Ontology Revision Definitions

We introduce the notion of backwards-compatibility to describe revisions that include all terms defined in the previous version and have the same intended meanings for them, although the axiomatizations may be different. This indicates that reasoners can safely assume that descriptions that commit to the old version also commit to the revision.

Definition 3.27 *An ontology O_2 is backwards-compatible with an ontology O_1 iff every intended model of O_1 is an intended model of O_2 and $V_1 \subseteq V_2$.*

Since the definition of backwards-compatible depends on knowledge of the intended models of an ontology, it cannot be computed automatically, instead it must be specified by an ontology's author. This is driven by the fact that ontologies only specify a theory partially, and that the intended meaning of a term may change even though the ontology's theory remains the same. Since the ontology can only restrict unintended models, there is no way to formally describe the intended models of an ontology. For example, if an ontology with a rather sparse axiomatization changed the term *Chair* to mean something you sit on as opposed to the head of a department, then if no relations or rules needed to be changed, any reasoning agent would be unaware that the term means different things in different versions. Thus backwards-compatibility must be indicated in an ontology definition. However, syntactic compatibility, such as whether $V_1 \subseteq V_2$, can be checked automatically, and when backward compatibility is specified, syntactic compatibility should be verified.

We will refine Definition 3.15 to include the concepts of an ontology revising another ontology and for an ontology to be backwards-compatible with older versions.

Definition 3.28 *Given a logic $\mathcal{L}$, an ontology is a five-tuple $\langle V, A, E, P, B\rangle$, where the vocabulary $V \subset S_P$ is some subset of the predicate symbols, the axioms $A \subset W$ are a subset of the well-formed formulas, $E \subset \mathcal{O}$ is the set of ontologies extended by O, $P \subset \mathcal{O}$ is the set of prior versions of the ontology, and $B \subset P$ is the set of ontologies that O is backwards compatible with.*

Definition 3.15 is a special case of this definition, where $P = \emptyset$ and $B = \emptyset$. All of the definitions from Section 3.3 still hold, although the five-tuple structure should be substituted for the three-tuple one where necessary.

We will also name two special cases of ontologies.

Definition 3.29 *A top-level ontology is an ontology $O = \langle V, A, E, P, B\rangle$, where $E = \emptyset$.*

Definition 3.30 *A basic ontology is an ontology $O = \langle V, A, E, P, B\rangle$, where $E = P = B = \emptyset$.*

Thus top-level ontologies are ontologies that have no ancestors; they are at the top of the ontology hierarchy. Every ontology must have at least one top-level ontology as an ancestor. Basic ontologies are top-level ontologies that have no prior versions.

Note that backwards-compatibility does not require that the revision contains a superset of the axioms specified by the original version. This allows axioms to be moved to a more general included ontology if needed.

3.4.3 Compatible Ontology Perspectives

Given the new definition of ontology, we can define a method of integration that incorporates backward-compatibility.

Definition 3.31 *Given a set of ontologies $\mathcal{O} = \{O_1, O_2, \ldots, O_n\}$ where $O_i = \langle V_i, A_i, E_i, P_i, B_i \rangle$, then a compatible ontology perspective based on ontology O_i is:*

$$\begin{aligned} COP_i(R) &= A_i \cup \bigcup_{\{j \mid O_j \in anc(O_i)\}} A_j \cup \bigcup_{\{r \in R \mid C(r) = O_i \vee C(r) \in anc(O_i)\}} K(r) \\ &\cup \bigcup_{\{r \in R \mid C(r) \in B_i\}} K(r) \cup \bigcup_{\{r \in R \mid \exists j, O_j \in anc(O_i) \wedge C(r) \in B_j\}} K(r) \end{aligned}$$

Like extended ontology perspectives, this method creates perspectives based upon different ontologies. Each perspective contains the axioms of its basis ontology, the axioms of its ancestors, and the assertions of all resources that commit to the basis ontology or one of its ancestors. However, these perspectives also include the assertions of resources that commit to any ontologies with which the basis ontology is backwards-compatible, and those of any resources that commit to ontologies that the base's ancestor ontologies are backwards-compatible with.

It should be mentioned that this method does not ensure that the perspective is logically consistent. The word compatible is used here in the sense of backward-compatibility, as defined in Section 3.4.2. The problem of inconsistency is discussed in Section 3.6.

As with extended ontology perspectives, a desirable property of compatible ontology perspectives is that a perspective based on an ontology entails all of the sentences that are entailed by any perspectives based on ancestors of the ontologies. We will show that compatible ontology perspectives satisfy this property.

Theorem 3.32 *Given two ontologies O_1 and O_2 such that $O_2 \in anc(O_1)$, if $COP_{O_2}(R) \models \phi$, then $COP_{O_1}(R) \models \phi$.*

Proof Let $O_1 = \langle V_1, A_1, E_1 \rangle$ and $O_2 = \langle V_2, A_2, E_2 \rangle$. We will prove the theorem by showing that $COP_{O_1}(R)$ is a superset of each of the parts of $COP_{O_2}(R)$. Since compatible ontology perspectives build on extended ontology perspectives, the proofs for the first three sets are identical. Since $O_2 \in anc(O_1)$, then every r such that $C(r) \in B_2$ is also in the fifth set of COP_{O_1}. Finally, since $anc(O_1) \supset anc(O_2)$, then for each r such that $O_j \in anc(O_2) \wedge C(r) \in B_j$, then also $O_j \in anc(O_1) \wedge C(r) \in B_j$. Therefore, the fifth set of COP_{O_1} subsumes that of COP_{O_2}. Since all sets that form COP_{O_2} are subsets of the sets that form COP_{O_1}, $COP_{O_1}(R) \supseteq COP_{O_2}(R)$. Since FOL is monotonic, if $COP_{O_2}(R) \models \phi$, then $COP_{O_1}(R) \models \phi$.

Also, if no ontologies revise any other ontologies, then compatible ontology perspectives are equivalent to extended ontology perspectives.

Theorem 3.33 *Given a set of ontologies $\mathcal{O} = \{O_1, O_2, \ldots, O_n\}$, where $\forall O_i \in \mathcal{O}, O_i = \langle V_i, A_i, E_i, \emptyset, \emptyset \rangle$, then $COP_i(R) = EOP_i(R)$.*

Proof If we substitute $\emptyset$ for B_i in Definition 3.31, then the set of $r \in R$ such that $C(r) \in B_i$ is empty because there are no r such that $C(r) \in \emptyset$. Therefore, the corresponding union is $\emptyset$. Likewise, the set of $r \in R$ such that $O_j \in anc(O_i) \wedge C(r) \in B_j$ must be empty, and the corresponding union is $\emptyset$. Thus the definition reduces to:

$$\begin{aligned}
COP_i(R) &= A_i \cup \bigcup_{\{j|O_j \in anc(O_i)\}} A_j \cup \bigcup_{\{r \in R|C(r)=O_i \vee C(r) \in anc(O_i)\}} K(r) \cup \emptyset \cup \emptyset \\
&= A_i \cup \bigcup_{\{j|O_j \in anc(O_i)\}} A_j \cup \bigcup_{\{r \in R|C(r)=O_i \vee C(r) \in anc(O_i)\}} K(r) \\
&= EOP_i(R)
\end{aligned}$$

Technically, the Semantic Web should not allow ontologies to arbitrarily revise other ontologies. Unlike, ontology extension, revision implies that a change has been authorized by the ontology's owner. Potential mechanisms for ensuring this include requiring older versions to point to their revisions, requiring revisions to be placed in the same directory of the same server as the ontology being revised, or to be signed by the same entity.

3.5 Ontology Divergence

As discussed earlier, an important aspect of this framework is that interoperability is achieved through ontology reuse. That is, the preferred method of ontology development is to extend existing ontologies and create new definitions only when existing definitions are unsuitable. In this way, all concepts are automatically integrated. However, when there is concurrent development of ontologies in a large, distributed environment such as the Web, it is inevitable that new concepts will be defined when existing ones could be used. Even when ontology authors have the best intentions, they may be unaware of similar efforts to describe the same domain, and their ontologies may be widely used by the time the problem is noticed. As a result there will be a tendency for the most specific ontologies to diverge and become less interoperable. In these situations, occasional manual integration of ontologies is needed.

This section discusses the types of semantic heterogenity that may occur in ontologies and presents a method for resolving ontology divergence within the framework presented earlier in this chapter. The ideas described here are a refinement of those presented in an earlier paper [52].

3.5.1 Domain Differences

The divergence of ontologies increases the semantic heterogeneity (see Section 2.4) of the Semantic Web. However, the use of first-order logic as our model results in a more restricted set of possible differences than those typically described by work in database schema integration. Wiederhold [91] describes four types of domain differences, which we paraphrase here:

context: a term in one domain has a completely different meaning in another

terminology: different names are used for the same concepts

scope: similar categories may not match exactly; their extensions intersect, but each may have instances that cannot be classified under the other

encoding: the valid values for a property can be different, even different scales could be used

Each of these differences can be resolved within our semantic web framework. Context differences are due to polysemous terms, and are handled by treating terms in each ontology as distinct. The other differences require the use of articulation axioms [19, 28], which are similar in purpose to lifting rules [46, Section 3.2]. An articulation axiom is simply an axiom that describes how to relate terms from two different ontologies. We will now demonstrate how to resolve the domain differences described above using axioms.

Terminological differences are synonyms, and as such can be handled using the equivalence idiom described in Section 3.3.2. For example, if it was determined that *Employee* in O_{kmart} meant the same thing as *StaffMember* in $O_{walmart}$, then the articulation axiom would be:

$$O_{kmart} : Employee(x) \leftrightarrow O_{walmart} : StaffMember(x)$$

Scope differences require mapping a category to the most specific category in the other domain that subsumes it. Thus, if we knew that every *FighterPilot* in O_{af} is a *JetPilot* in O_{faa}, then we would create the articulation axiom:

$$O_{af} : FighterPilot(x) \rightarrow O_{faa} : JetPilot(x)$$

Encoding difference are somewhat trickier. The problem is that different sets of values are used to describe the same data. These sets may have different cardinalities or may be infinite. An example of value sets with different cardinalities may be two rating schemes for movies. One scheme uses $\{$*Poor*,*Fair*,*Excellent*$\}$ while the other uses integers 1-5. In this case, individual values could be mapped as in:

$$O_{siskel} : Rating(x, Excellent) \leftrightarrow O_{ebert} : Rating(x, 5)$$

Other differences may be due to different units, such as meters versus feet. Articulation axioms to resolve these sorts of encodings would require the use of arithmetic functions, as in:

$$O_{english} : Foot(x, l) \rightarrow O_{metric} : Meter(x, *(l, 0.3048))$$

Note that arithmetic functions are simply functions whose domains range over integers or real numbers, and thus do not require any special treatment in first-order theory. However, such functions can be problematic in reasoning algorithm implementation. For example, unit conversion may introduce inaccuracies due to floating point arithmetic and rounding. This can get compounded if ontologies have rules for translating both ways. For example, if a reasoner translated 3 feet to 0.914

meters, it better not then apply the opposite rule and get a length of 2.999 feet as well. Such a process could go on ad infinitum. An even more difficult encoding difference is due to different textual representations. Consider "Smith, John" versus "John Smith." An articulation axiom to establish name correspondences in general would require a function that can take the last-name-first form and convert it to the first-name-first form. Although this is easy in theory, in practice it requires a large list of pre-defined functions or a complex language for defining functions.

3.5.2 Resolving Ontology Divergence

Ontology integration typically involves identifying the correspondences between two ontologies, determining the differences in definitions, and creating a new ontology that resolves these differences. The process for aligning ontologies can be performed either manually or semi-automatically. Chimaera [69] and PROMPT [75] are examples of tools that help users to align ontologies. However, it is important to note that simply creating a new integrated ontology does not solve the problem of integrating information on the Web. When the web community has synthesized the ontologies (that is, other web pages and ontologies come to depend on them), all of the dependent objects would have to be revised to reflect the new ontology. Since this would be an impossible task, we instead suggest three ways to incorporate the results of an ontology integration effort, each of which is shown in Figure 3.6. In this figure, we assume that O_1 and O_2 are two ontologies which have some domain overlap and need to be integrated.

In the first approach, we create a third ontology, called a mapping or articulation ontology, that can translate the terminologies of the two ontologies. In the example, the mapping ontology is O_M. In order to map the terminologies, O_M must extend

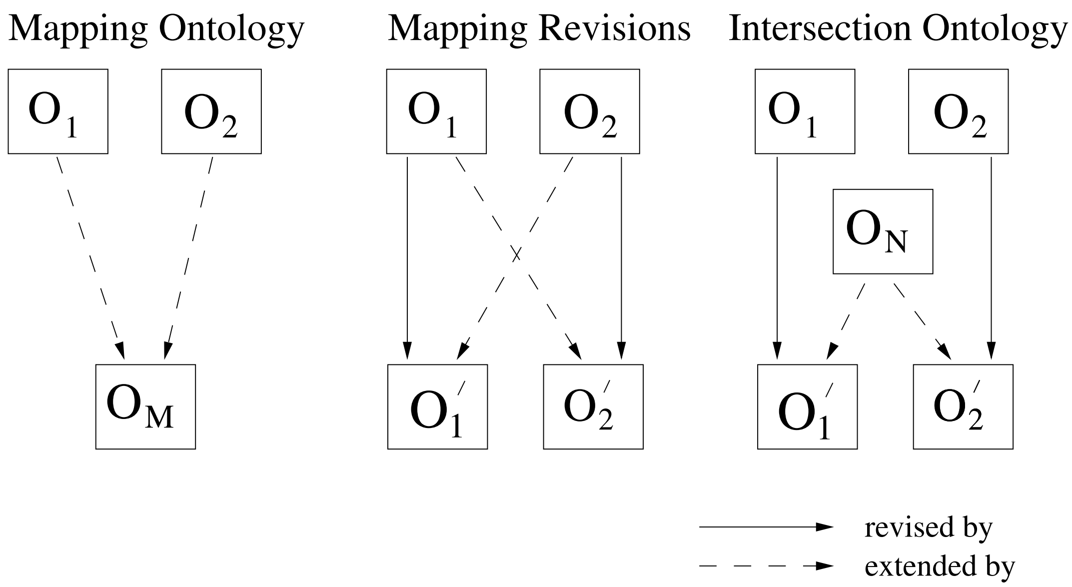

Figure 3.6: Methods for resolving ontology divergence.

both O_1 and O_2, and provide a set of articulation axioms T as described above. Note that O_M does not add any vocabulary terms, thus $O_M = \langle \emptyset, T, \{O_1, O_2\}, \emptyset, \emptyset \rangle$. The advantage of a mapping ontology is that the domain ontologies are unchanged; thus, it can be created without the approval of the owners of the original ontology. However, since it is like any other ontology, it can be made publicly available and used by anyone who would like to integrate the ontologies. The disadvantages are that the integration only occurs in the perspective that is based on the mapping ontology, if the source ontologies are revised then a new mapping ontology must be created, and a set of articulation axioms are needed for each additional ontology that covers the domain.

Another approach to implementing integration is to revise each ontology to include mappings to the other. First, we create a new version of each ontology, called a mapping revision. Each revision extends the original version of the other ontology and includes a set of articulation axioms, allowing it to translate the terms from that ontology. Since each revision leaves the original vocabulary unchanged, and (assuming the articulation axioms are correct) does not change the intended models, it is backward-compatible with the original version. Thus, in the example where O_1' is the mapping revision of O_1 and O_2' is the mapping revision of O_2, if T_1 is the set of articulation axioms from the vocabulary of O_2 to that of O_1 and T_2 is the set of articulation axiom from the vocabulary of O_1 to O_2, then $O_1' = \langle V_1, A_1 \cup T_1, \{O_2\}, \{O_1\}, \{O_1\}\rangle$ and $O_2' = \langle V_1, A_2 \cup T_2, \{O_1\}, \{O_2\}, \{O_2\}\rangle$. This ensures that perspectives based on O_1' and O_2' will integrate resources that commit to O_1 and O_2. The advantage of this approach is that the articulation axioms are inserted into the latest versions of the ontologies, ensuring that they will apply to later backward-compatible revisions. The main disadvantage is that due to the nature of revision (see page 69), it can only be performed by the owners of the original ontologies.

A common disadvantage of the mapping ontology and mapping revision approaches is that they ignore a fundamental problem: the overlapping concepts do not belong in either domain, but are more general. The fact that two domains share the concept may mean that other domains will use it as well. If this is so, then each new domain would need a set of articulation axioms to map it to the others. Obviously this can become unwieldy very quickly. A more natural approach is to merge the common items into a more general ontology, called an intersection ontology, which is then extended by revisions to the domain ontologies. First, we create a set of terms and axioms that standardize the commonalities between O_1 and O_2, referred to as V_N and A_N, respectively. We then create a new ontology O_N, where $O_N = \langle V_N, A_N, \emptyset, \emptyset, \emptyset\rangle$. Then we determine a set of articulation axioms T_1 and T_2 that translate from O_N to O_1 and O_2, respectively. When combined with A_N, these will allow us to conclude some formulas already in A_1 and A_2; we will refer to the sets of these formulas as D_1

and D_2. Formally, $\phi \in D_1$ iff $\phi \in A_1$ and $A_N \cup T_1 \models \phi$ (similarly for D_2). Now we can define the revised ontologies O'_1 and O'_2. $O'_1 = \langle V_1, A_1 - D_1, \{O_N\}, \{O_1\}, \{O_1\}\rangle$ and $O'_2 = \langle V_2, A_2 - D_2, \{O_N\}, \{O_2\}, \{O_2\}\rangle$. Note that as with the mapping revisions approach, the revised ontologies retain the vocabulary and do not change the intended models, so they can be backward-compatible.

3.6 Inconsistency

If the logical language used by the Semantic Web is rich enough to express inconsistency, then an inconsistency within a single resource or one that exists between a pair of resources will result in one or more perspectives that are inconsistent. For example, since first-order logic is monotonic, if $K(r_1) = \{A\}$ and $K(r_2) = \{\neg A\}$ then any perspective which contains both r_1 and r_2 is inconsistent. As is well known, such an inconsistency trivializes first-order logic, allowing anything to be proven. However, the distributed nature of the Web makes it impossible to guarantee that inconsistencies will not be stated. This results in another principle of the Semantic Web:

Principle 3.34 *An inconsistency due to a single resource or a pair of resources should not make an entire perspective inconsistent.*

Although perspectives solve some of the problems of handling distributed ontologies, logical inconsistency is still a danger. A perspective can become inconsistent in three ways: if the basis ontology is inconsistent with one of its ancestors, if a resource is inconsistent with the basis ontology (or an ancestor), or if two resources are inconsistent with each other. If an ontology is designed carefully, then inconsistency between it and one of its ancestors can be prevented. We will require that a valid ontology must be consistent with all of its ancestors. When a resource commits to an ontology, it implicitly agrees to the intended models of the ontology. If the resource includes an assertion that is inconsistent with the ontology, then this agreement is violated. Therefore, it is safe to assume that such resources are invalid and can be excluded from all perspectives. However, there is still the problem that two resources which commit to the same ontology could be inconsistent. If so, any perspective that included this ontology's resource would be inconsistent. Due to the dynamic nature of resources, a resource's validity should not depend on any other resource. However, given the distributed nature of the Web, it is impossible to prevent two resources from contradicting each other. Thus, perspectives created using the approaches described above will likely become inconsistent, and cannot be easily fixed. Indeed, this could even be the form of a new, insidious, and incredibly simple denial of service attack: publish an inconsistency and watch the Semantic Web grind to a halt. Clearly, there must be a way to prevent or resolve potential inconsistencies.

A common way to handle inconsistencies is through some form of nonmonotonicity. In nonmonotonic logics, certain statements are considered defaults, and are only true if it they are not inconsistent with more explicit information. Often, an implicit assumption in these theories is that the most recent information is correct and that it is prior beliefs that must change. On the Web, this assumption cannot be made; if anything, more recent information that conflicts with prior beliefs should be approached with skepticism. Additionally, inconsistencies on the Web will often be due to fundamental disagreements, and thus neither statement can be considered the "default."

The (primarily philosophical) field of belief revision [38] focuses on how to minimize the overall change to a set of beliefs in order to incorporate new inconsistent information. A representation of an agent's knowledge at some point in time is called an epistemic state, and a change of knowledge leads to a new epistemic state via an epistemic change. The three types of epistemic changes are expansions, revisions, and contractions. An expansion adds an assertion that is consistent with the existing epistemic state. A revision adds an assertion that is inconsistent with existing beliefs, and requires that some knowledge be retracted in order to determine the new epistemic state. Finally, a contraction removes an assertion, which may lead to the removal of other assertions that depend on it. Gardenförs [38] presents a series of postulates describing epistemic changes. An important criterion is that of minimal change, that is, the only changes made to the epistemic state are those required to accommodate the new information. In the case of revision, this may require choosing between equally adequate alternatives. In such cases, the relative epistemic entrenchment of the beliefs (which determines how important they are), may be used to choose an epistemic state. However, on the Semantic Web, it is unclear how the epistemic entrenchment of an assertion should be determined. Furthermore, it is unclear that maintaining a single consistent set of beliefs makes sense in a distributed knowledge system.

The chief problem with nonmonotonic logic and theories of belief revision is choosing which set of assertions should constitute the "beliefs" of the Semantic Web. Assumption-based truth maintenance systems (ATMSs) [21] present an alternative. In an ATMS, multiple contexts are maintained, where each context represents a set of consistent assumptions. Thus it is is possible to consider the truth of a formula with respect to a particular context, or to determine the set of contexts in which a formula is true. If we assume that each ontology and resource must be internally consistent, then there can be contexts that assume that each is individually true. More complex contexts can be formed from these by assuming that multiple ontologies or resources are true at the same time. However, if there are n ontologies and resources, then there could be as many as 2^n possible contexts. Although contradictions can automatically be propagated from simpler contexts to the complex contexts that contain them, management of contexts would be a serious problem on the Semantic Web. The Web already contains over a billion resources (web pages), and many more resources are added every day. Each new resource would have to be compared with every existing context to determine which new consistent contexts can be formed.

A different solution is to limit the language so that it is not possible to express logical inconsistencies. In first-order logic, this can be done by omitting negation.[1] Other logics, particularly description logics, include features such as cardinality constraints and the specification of disjoint classes, which can lead to inconsistency. The main argument against limiting the language to prevent logical inconsistency is that inconsistency can be a sign that two theories should not be combined. Still, the advantage of limiting the language is that it does not have the computational or philosophical problems associated with the other methods.

Unlike the previous sections, we do not suggest a solution to the problem of inconsistency here. We have discussed the relative benefits of various alternatives, but believe that only future research will determine the best choice for the Semantic Web. In Section 4.1, we will discuss the choice made for the SHOE language.

[1]Here we mean only omission of the logical operator, and not of other logical connectives that can be rewritten by using negation, such as implication.

3.7 Scalability

Throughout this chapter, we have used first-order logic as the basis for our discussion of the Semantic Web. However, sound and complete reasoning in first-order logic is intractable, meaning that there is no polynomial-time algorithm that solves the problem. Thus, first-order logic systems will not scale to the quantity of formulas that would be expected on the Semantic Web. So then how can the problem of scalability be handled in a practical setting?

One approach is to use reasoning methods that are not sound and complete. Resource-bounded reasoning algorithms (that limit the computation time, number of steps, etc.) are quite common and would be applicable for many Semantic Web applications. In many cases, it is not necessary to know all of the answers on the Semantic Web, only a handful of correct ones will suffice. Given the extent of the Web, it is unlikely that any reasoner will have access to all of the assertions, so it is improbable that even one which used a sound and complete algorithm would be truly complete in the global sense.

Another approach to scalability is to reduce the expressivity of the language. This has been an important direction for the knowledge representation community which has tried to characterize the computational complexity of languages with various features. Starting with Brachman and Levesque [11], the complexity of different forms of description logics has been computed, and languages have been developed that attempt to maximize expressivity while minimizing complexity. Even so, subsumption is intractable in many description logics.

An alternative to description logics is to use Horn logic. It has been shown that although Horn-logic and the most common description logics can express things the other cannot, neither is more expressive than the other [8]. Thus the relative advantages of the two languages depend on the kinds of constructs viewed as most useful to the Semantic Web. In Section 3.6, we discussed the problems inherent in languages that include negation or other features that may lead to inconsistency; most description logics face these problems due to the presence of cardinality restrictions. Horn logic on the other hand can not be logically inconsistent. Furthermore, if we restrict the language to datalog, which is a minor variant of Horn logic, then polynomial reasoning algorithms such as the magic sets technique can be used.

As in Section 3.6, we do not provide a solution to scalability problem here. This is another difficult issue, and only future use of the Semantic Web will determine the right combination of language features and query methods. In Section 4.1 we will explain the choice made for the SHOE language, and in Section 5.2 will discuss the use of reasoning systems with different inferential capabilities.

3.8 Semantic Web Queries

The design of a semantic web language requires consideration of how the language will be used. The Semantic Web can be used to locate documents for people or to answer specific questions based on the content of the Web. These uses represent the document retrieval and knowledge base views of the Web.

The knowledge base view uses the logical definition of queries: a query is a formula with existentially quantified variables, whose answers are a set of bindings for the variables that make the formula true with respect to the knowledge base. But what is a knowledge base in the context of the Semantic Web? In order to resolve a number of problems faced by the Semantic Web we have extensively discussed means of subdividing it. Theoretically, each of these perspectives represents a single model of the world, and could be considered a knowledge base. Thus, the answer to a semantic web query must be relative to a specific perspective.

Consider the set of ontologies and resources presented in Figure 3.7. There are three compatible ontology perspectives generated from this data: $COP_G(R)$, $COP_U(R)$, and $COP_F(R)$. Based on Definition 3.31, different ontologies and resources appear in each perspective. For example, $COP_G(R)$ includes the axioms from O_G and the knowledge from r_1 and r_2. It does not include r_3 or r_4 because these commit to other ontologies. $COP_U(R)$ includes the axioms from O_U and, because O_G is an ancestor of O_U, those of O_G. It also includes the resources r_1, r_2, and r_3, which commit to these ontologies. On the other hand, $COP_F(R)$ includes axioms from O_F and O_G, and the resources r_1, r_2, and r_4. As a result, the answer to any particular query depends on which perspective it is issued against. As shown in Figure 3.8, the answer to $Person(x)$ in $COP_G(R)$ is just $\{bob\}$ because from this perspective the axioms and resources of O_U are irrelevant. However, in $COP_U(R)$, the answer is $\{bob, kate\}$ because we have the axiom from O_U that tells us every ***Chair*** is a ***Person***. Also note that in $COP_F(R)$, the answer is $\{bob\}$ because O_F includes O_G. When we ask a query such as $Chair(x)$, then the variety in answers is even greater. In $COP_U(R)$ the answer is $\{kate\}$ while in $COP_F(R)$ it is $\{recliner29\}$. This is because the perspectives decontextualize the term *Chair* differently. Also note that in $COP_G(R)$, the query is ill-formed with respect to the ontology that serves as the basis of the perspective (i.e., the ontology does not include *Chair* in its vocabulary), and thus there can be no answers.

From the point of view of most agents, the Semantic Web can be seen as read-only. This is because most web pages can only be updated by their owners via the file mechanisms on their servers. A file is either saved or unsaved, and only saved files are available via HTTP. Thus, the update of a file becomes a single transaction

$$
\begin{aligned}
O_G = \quad & \langle\{Thing, Person, Object\}, \\
& \{Person(x) \rightarrow Thing(x), \\
& Object(x) \rightarrow Thing(x)\}, \\
& \emptyset, \\
& \emptyset, \\
& \emptyset\rangle \\
O_U = \quad & \langle\{Chair, Person, Object\}, \\
& \{Chair(x) \rightarrow O_G : Person(x), \\
& Person(x) \leftrightarrow O_G : Person(x), \\
& Object(x) \leftrightarrow O_G : Object(x)\} \\
& \{O_G\}, \\
& \emptyset, \\
& \emptyset\rangle \\
O_F = \quad & \langle\{Chair, Person, Object\}, \\
& \{Chair(x) \rightarrow O_G : Object(x), \\
& Person(x) \leftrightarrow O_G : Person(x), \\
& Object(x) \leftrightarrow O_G : Object(x)\} \\
& \{O_G\}, \\
& \emptyset, \\
& \emptyset\rangle \\
C(r_1) = \quad & O_G \\
K(r_1) = \quad & \{Person(bob)\} \\
C(r_2) = \quad & O_G \\
K(r_2) = \quad & \{Object(sofa52)\} \\
C(r_3) = \quad & O_U \\
K(r_3) = \quad & \{Chair(kate)\} \\
C(r_4) = \quad & O_F \\
K(r_4) = \quad & \{Chair(recliner29)\}
\end{aligned}
$$

Figure 3.7: Example ontologies and resources.

	Perspective		
Query	$COP_G(R)$	$COP_U(R)$	$COP_F(R)$
$Person(x)$	bob	$bob, kate$	bob
$Object(x)$	$sofa52$	$sofa52$	$sofa52, recliner29$
$Chair(x)$	n/a	$kate$	$recliner29$

Figure 3.8: Query answers for different perspectives.

and many issues that are important to databases, such as concurrency and serializability are not significant. Although web pages usually have a single writer, there are many web pages that change frequently. If the agent reaccesses such pages in the middle of the query, it may be presented with a different set of assertions. For these reasons, it is recommended that for the duration of each query, reasoning systems cache the assertions of all resources used in the query.

Document retrieval queries can locate a document that represents a concept, which may or may not be partially defined, or locate a document that has specified metadata. Here, metadata is data about the document itself, such as its author or modification date. When the domain of the language includes resources, then the knowledge base view subsumes the the document retrieval view. That is, we can specify the relationship between a document and the concept that it represents, and we can describe both the document and the concept independently.

3.9 Summary

In this chapter, we have gradually developed a formal model for describing the Semantic Web. The foundation for this model is first-order logic, but we found that we needed ontologies to represent common background knowledge and provide reusable vocabularies. We then presented a method of partitioning ontologies and resources to ensure that only those that shared the same context were integrated. We extended this model with the ability to specify ontology inclusion, so that content providers could describe their own information needs while still reusing existing ontologies. This allows us to increase the integration of distributed resources, as is done with extended ontology perspectives. We further extended the model to deal the problem of ontology evolution, and discussed the issue of backward-compatibility.

This resulted in compatible ontology perspectives which can integrate resources that commit to any ancestors of an ontology as well as resources that commit to forward-compatible prior versions of the ontology. We discussed how extension alone would be insufficient for integrating resources in a distributed ontology environment, and discussed the problem of ontology divergence. Another important issue in distributed environments is the inability to preserve integrity constraints and the likelihood of global inconsistency. The size of the Web makes scalability an important issue and possible approaches were also discussed. Finally, we talked about queries on the Semantic Web, and how they depend on perspectives.

Chapter 4

The SHOE Language

In the previous chapter, we examined the problem of the Semantic Web and developed a framework that handles some of these problems. In this chapter, we will present SHOE, an actual Web language based on these concepts. SHOE [66, 48], which stands for Simple HTML Ontology Extensions, was originally developed by the PLUS Group at the University of Maryland in 1995. Since then, the PLUS Group has refined the language and experimented with its use.

SHOE combines features of markup languages, knowledge representation, datalog, and ontologies in an attempt to address the unique problems of semantics on the Web. It supports knowledge acquisition by augmenting the Web with tags that provide semantic meaning. The basic structure consists of ontologies, which define rules that guide what kinds of assertions may be made and what kinds of conclusions may be drawn from these assertions, and instances that make assertions based on those rules. As a knowledge representation language, SHOE borrows characteristics from both predicate logics and frame systems.

4.1 Design Rationale

Although we have developed a formal model in Chapter 3, an actual language needs to address issues such interoperability with existing technology, language expressivity, performance, and usability. In this section, we will make a number of choices regarding these issues and explain the rationale behind these choices. Note that in many cases there is no clear right answer, so some choices are based on intuition and experimentation.

First, we will address the issue of syntax. Although the standard languages of knowledge representation are Lisp-based and Prolog-based, the Web is dominated by HTML and XML. Since the Web community is much larger than the knowledge representation one, there is a strong reason to choose a Web-like syntax over a traditional knowledge representation one. Such a language could embedded in HTML documents, allowing it to be added to legacy web pages with minimal hassle, or it could be embedded in XML documents. An XML syntax can be analyzed and processed using the Document Object Model (DOM), which can be parsed and manipulated by a number of free and commercial libraries, providing a strong foundation upon which developers can build Semantic Web tools. Additionally, software which is XML-aware, but not SHOE-aware, can still use the information in more limited but nevertheless powerful ways. For example, some web browsers are able to graphically display the DOM of a document as a tree, and future browsers will allow users to issue queries that will match structures contained within the tree. A second reason for using an XML syntax is that SHOE documents can then use the XSLT stylesheet standard [18] to render SHOE information for human consumption, or translate it into other XML formats. We chose to give SHOE both an HTML and an XML syntax, thus allowing it to be used by the entire web community, regardless of whether or not they have migrated to XML.

In Chapter 3 we described the need for ontologies, and presented a model of how ontologies could be used on the Semantic Web. Ontologies are a critical component of SHOE, although they are formulated somewhat differently than the five-tuples of Definition 3.28. First, a SHOE ontology has both an identifier and a version number, where it is assumed that all ontologies with the same identifier are different versions of the same ontology. This approach avoids the problem of having to list every previous version of the ontology, since these can be determined by comparing identifiers and version numbers. Additionally, backwards-compatible revisions can be specified by version number only. In Section 4.3, we will explicitly describe how SHOE ontologies relate to the framework presented in Chapter 3.

SHOE has a feature for ontology inclusion, and the included ontology is specified by a combination of the ontology's identifier and version number. In a distributed environment such as the Web, few agents will know every ontology. To help agents locate an unfamiliar ontology, SHOE also allows the URL of an included ontology to be specified. To handle potential name conflicts between ontologies, every SHOE ontology has its own namespace, and a special prefixing mechanism is used to refer to the components of another ontology.

Since SHOE is meant to be used by ordinary web authors, it is unrealistic to expect them to provide complex first-order logic formulas to describe their content. SHOE follows the strategy of the OKBC API [17] and the Ontolingua language [42], which both have first-order logic based foundations, but provide frame-based idioms for convenience. A frame-based paradigm tends to be easier to use, because it has similarities to object-oriented programming languages and databases, which have become quite popular for software development. SHOE has categories (commonly called classes in the knowledge representation literature), which can be thought of as frames, and relations, which determine the slots of those frames. Categories can have supercategories, with the semantics of an *isa* link. The arguments of SHOE relations are typed, and SHOE provides some basic data types for this purpose. These basic data types can be used to check the syntax of values and to provide an interpretation for the value. To handle the common case of synonyms, SHOE also has an aliasing feature. Finally, for advanced ontology designers, SHOE provides additional axioms called inference rules. All of SHOE's features have equivalent first-order logic expressions, which ensure that the framework from Chapter 3 still applies.

As discussed in Section 3.7, the kinds of axioms that can be expressed in a language determines its scalability. Recall that sound and complete reasoning in first-order logic is intractable and thus of little use on real-world problems such as those presented by the Semantic Web. It is expected that on the Web, facts will vastly outnumber axioms. Since deductive database research operates under the same assumptions, we chose to base SHOE's semantics on datalog, and make use of the algorithms and systems developed for it. Note that the categories, relations, and alias features can all be expressed in datalog, so the only consequence of this decision is that we must restrict the axioms to Horn clauses. In Chapter 7, we will discuss some Semantic Web languages that have made different choices.

SHOE associates knowledge with resources by declaring instances. These instances contain assertions about themselves and other instances, and the assertions may describe membership in particular categories, or relations that hold. Every instance commits to at least one ontology, which defines the categories and relations used.

Every instance must have a key, which is used to reference it, but creating and assigning such keys can be problematic in distributed environments. However, URLs provide a good mean for identifying resources on the internet, and can be used as the basis for forming instance keys. It is assumed that each key identifies exactly one entity, but no assumptions are made about whether two distinct keys might identify the same entity. This is because many different URLs could be used to refer to the same resource (due to the facts that a single host can have multiple domain names and operating systems may allow many different paths to the same file). To solve these problems in a practical setting, a canonical form can be chosen for the URL; an example rule might be that the full path to the file should be specified, without operating systems shortcuts such as '~' for a user's home directory. Even then, there are still problems with multiple keys possibly referring to the same conceptual object. Thus, this solution ensures that the system will only interpret two objects as being equivalent when they truly are equivalent, but ensuring that two object references are matched when they conceptually refer to the same object is an open problem.

Finally, we must address the issue of potential inconsistency. As discussed in Section 3.6, if inconsistency is not managed, then any theory formed by combining resources can be easily trivialized, degrading the usefulness of the language. Although it is possible to create perspectives that only include maximally consistent sets of ontologies and resources, the process is complex and inefficient. Instead, we chose to keep SHOE easy to understand and implement, and have carefully designed the language to eliminate the possibility of contradictions between agent assertions. SHOE does this in four ways:

1. SHOE only permits assertions, not retractions.

2. SHOE does not permit logical negation.

3. SHOE does not allow relations to specify a cardinality, and thus limit how many relation assertions of a particular kind can be made for any single instance.

4. SHOE does not permit the specification of disjoint classes.

Clearly, this restricts the expressive power of the language; it could be argued that without these features, agents cannot recognize when resources are inherently incompatible and should never be combined. While this is true, accidentally combining a few resources incorrectly and drawing a few false conclusions is more appealing than having to check every resource for inconsistency with every other resource, or worse, accidentally combining two inconsistent resources to create a trivialized theory. However, research into pragmatic ways to handle semantic web inconsistencies is deserving of future work.

We have explained the basic decisions in the design of the SHOE language. In the next section, we describe the resulting language in detail.

4.2 Language Description

This section describes the SHOE language, which provides a way to incorporate machine-readable semantic knowledge in World Wide Web documents. We will describe the syntax and semantics of ontologies, instances, and their components.

4.2.1 A Comment on Syntax

SHOE has two syntactical variations. The first syntax is an SGML application that extends the HTML syntax with additional semantic tags. This syntax can be used to embed SHOE in ordinary web documents. To indicate conformance with SHOE, HTML documents must include the following text in the HEAD section of the document:

```
<META HTTP-EQUIV="SHOE" CONTENT="VERSION=1.0">
```

The CONTENT of the meta-tag indicates that the document is compliant with version 1.0 of the SHOE language.

Sections 4.2.3 and 4.2.4 describe the elements of SHOE and present the remainder of the SGML syntax. The syntactic descriptions in these sections use a sans serif font to indicate key words of the language and *italics* to indicate that the author must supply a parameter or expression. Brackets ('[' or ']') are used to indicate optional portions of the syntax. Since SGML is not case-sensitive, the element and attribute names can appear in any case. Arbitrary white space is allowed between attributes within a tag and between tags. Also, the quotes around attribute values may be omitted if the value does not contain white space. A complete and concise specification of the syntax is given by the SGML DTD for SHOE, which is provided in Appendix A.1,

The second SHOE syntax is an XML application. While the SGML syntax allows SHOE to be easily embedded in the numerous existing HTML web pages, the XML syntax allows SHOE to leverage emerging web standards and technologies. Since XML is basically a subset of SGML, the XML syntax for SHOE is very similar to the SGML one. The SHOE XML DTD is presented in Appendix A.2.

The XML version of SHOE can either stand alone, or be included in another XML document. A stand-alone SHOE XML document must begin with the appropriate XML prolog:

```
<?xml version="1.0"?>
<!DOCTYPE shoe SYSTEM
   "http://www.cs.umd.edu/projects/plus/SHOE/shoe_xml.dtd">
```

The root element of this document must be shoe, and it must contain a version attribute with value “1.0”, as shown below:

```
<shoe version="1.0">
```

All SHOE elements must be between this tag and a closing </shoe> tag.

Alternatively, SHOE can be embedded in other well-formed XML documents. When documents combine different element sets, they must use XML namespaces [14] to prevent accidental name clashes. The simplest way to do this with SHOE is to set the default namespace within the shoe element.

```
<shoe xmlns="http://www.cs.umd.edu/projects/plus/SHOE/"
      version="1.0">
```

Note that most of the Web’s HTML is not well-formed XML. However, the Extensible HyperText Markup Language (XHTML) [89] provides a variation of HTML that is compatible. Thus, an HTML document could be converted to XHTML, and then SHOE can be added to it.

Although the syntax presented in sections 4.2.3 and 4.2.4 describe the elements of SHOE using the SGML syntax, most of it is still applicable to the XML variation. However, since XML is more restrictive, the following additional rules must be applied:

- All empty elements, i.e., elements which have no content and no end tag, must end with a '/>' instead of a '>'. Specifically, this applies to the USE-ONTOLOGY, DEF-CATEGORY, DEF-ARG, DEF-RENAME, DEF-CONSTANT, DEF-TYPE, CATEGORY, and ARG elements.

- No attribute minimization is allowed. In SGML attribute minimization allows the names of some attributes to be omitted and only their values to be specified.

In the SGML syntax, this is usually used to specify VAR within the subclauses of an inference rule (see page 105). In the XML syntax, the attribute name USAGE must be explicitly provided, e.g. USAGE="VAR" instead of VAR.

- Since XML is case-sensitive, all element and attribute names must be in lower case.

- All attribute values must always be quoted, including those which are numeric as well as the FROM and TO keywords.

4.2.2 The Base Ontology

The base ontology is the ultimate ancestor of all other ontologies. There is a one-to-one correspondence between versions of the SHOE language and versions of the base ontology, thus the version number of the META tag or the version attribute of the shoe element indicates which version of the base ontology is applicable.

The base ontology provides some fundamental categories, relations, and data types. It defines the categories Entity and SHOEEntity, where the latter is a subcategory of the former. SHOEEntity is the superclass of all classes defined in other SHOE ontologies. The relations name and description can be used to provide names and definitions for any instance. Finally, the base ontology defines four basic data types, which can constrain the values used in relation assertions. The data types are:

STRING A sequence of ASCII characters, including but not limited to letters, digits, and punctuation. Strings have an implicit alphabetical ordering.

NUMBER A floating-point numeric literal. These can be provided as integers, decimals, or in the standard exponential notation, i.e., they have the form:

```
[+|-] digit+ ['.' digit+] [(('e'|'E')[+|-]digit+)]
```

The standard ordering applies to all numbers.

DATE A date and time, based on RFC 1123. Values may have the following form:

```
WWW, DD MMM YYYY HH:MM:SS TZD
```

where WWW is a three letter abbreviation for the day of week, DD is the day of month, MMM is the three letter abbreviation of the month, YYYY is the four digit year, HH is the two digit hour (from 0 to 23), MM is the two digit minute, SS is the two digit second, and TZD is the time zone designator. Dates are ordered chronologically.

TRUTH **(Boolean)** A boolean value, either YES or NO, case-insensitive. For comparison purposes, NO is considered less than YES.

4.2.3 Ontology Definitions

SHOE uses ontologies to define the valid elements that may be used in describing instances. An ontology is stored in an HTML or XML file and is made available to document authors and SHOE agents by placing it on a web server. The ontology can include tags that state which ontologies (if any) are extended, and define the various elements of the ontology, such as categories, relations, and inference rules. Figure 4.1 shows an example of a SHOE ontology.

Each ontology has an identifier and a version number that uniquely defines it. Accidental reuse of ontology identifiers can be avoided by including the domain name of its author in this identifier. Ontologies with the same identifier but different version

```
<!-- Declare an ontology called "university-ont". -->
<ONTOLOGY ID="university-ont" VERSION="1.0">

<!-- Extend the general ontology, assign it the prefix "g." -->
   <USE-ONTOLOGY ID="general-ont" VERSION="1.0" PREFIX="g"
                 URL="http://www.ontology.org/general1.0.html">

<!-- Create local aliases for some terms -->
   <DEF-RENAME FROM="g.Person" TO="Person">
   <DEF-RENAME FROM="g.Organization" TO="Organization">
   <DEF-RENAME FROM="g.name" TO="name">

<!-- Define some categories and subcategory relationships -->
   <DEF-CATEGORY NAME="Faculty" ISA="Person">
   <DEF-CATEGORY NAME="Student" ISA="Person">
   <DEF-CATEGORY NAME="Chair" ISA="Faculty">
   <DEF-CATEGORY NAME="Department" ISA="Organization">

<!-- Define some relations; n-ary relations are also allowed -->
   <DEF-RELATION NAME="advises">
      <DEF-ARG POS="1" TYPE="Faculty">
      <DEF-ARG POS="2" TYPE="Student">
   </DEF-RELATION>

   <DEF-RELATION "hasGPA">
      <DEF-ARG POS="1" TYPE="Student">
      <DEF-ARG POS="2" TYPE=".NUMBER">
   </DEF-RELATION>

<!-- Define a rule: The head of a Department is a Chair -->
   <DEF-INFERENCE>
      <INF-IF>
         <RELATION NAME="g.headOf">
            <ARG POS="1" VALUE="x" USAGE="VAR">
            <ARG POS="2" VALUE="y" USAGE="VAR">
         </RELATION>
         <CATEGORY NAME="Department" FOR="y" USAGE="VAR">
      </INF-IF>
      <INF-THEN>
         <CATEGORY NAME="Chair" FOR="x" USAGE="VAR">
      </INF-THEN>
   </DEF-INFERENCE>

</ONTOLOGY>
```

Figure 4.1: An example university ontology.

numbers are considered to be different versions of the same ontology. An ontology is a revision of another ontology if it has the same identifier but a later version number.

A SHOE document may contain any number of ontology definitions. Many of the definitions within an ontology have an associated name, and these are collectively called named components. The named components are categories, relations, constants, and types. The names of these components are all subject to the same restrictions: they must begin with a letter, may only contain letters, digits, and hyphens; cannot contain whitespace, and are case-sensitive. There is a single namespace for all named components, and thus it is invalid for an ontology to define two components that have the same name.

In HTML documents, the ONTOLOGY element must be a subelement of the BODY element; in XML documents, it must be a subelement of the shoe element. Only the SHOE elements described in the rest of this section may be nested in the ONTOLOGY element. An ONTOLOGY element has the following form:

<ONTOLOGY ID="id"
VERSION="$version$"
[BACKWARD-COMPATIBLE-WITH="bcw_1 bcw_2 ...bcw_n"]
[DESCRIPTION="$text$"]
[DECLARATORS="dec_1 dec_2 ...dec_m"] >
$content$
</ONTOLOGY>

ID (mandatory) Specifies the ontology's identifier. This must begin with a letter, contain only letters, digits, and hyphens; and may not contain whitespace. Identifiers are case-sensitive.

VERSION (mandatory) Specifies the ontology's version number. Version numbers may only contain digits and dots.

BACKWARD-COMPATIBLE-WITH Specifies a whitespace-delimited list of previous versions that this ontology subsumes. Each bcw_i must be a valid ontology version number.

DESCRIPTION A short, human-readable description of the purpose of the ontology.

DECLARATORS Specifies a whitespace-delimited list of URLs for content resources the ontology has associated with itself. Ordinarily, an ontology cannot assert relationships or categorizations, only define the rules that govern such assertions. This mechanism allows an ontology to state that one or more resources contain important standard assertions associated with the ontology. See Section 4.2.4 for information on specifying assertions.

$content$ All of an ontology's definitions and extensions must appear between the open and close ONTOLOGY tags. This includes the USE-ONTOLOGY, DEF-CATEGORY, DEF-RELATION, DEF-RENAME, DEF-INFERENCE, DEF-CONSTANT, and DEF-TYPE elements. The elements are described in the rest of this section.

Extending An Existing Ontology

An ontology can extend one or more ontologies so that it may use their elements and sanction their rules. The extended ontology is indicated by its identifier and version number. If an agent cannot locate the ontology, it is free to ignore it. Although it is expected that there will be certain standard ontologies which are known to all agents, it is good practice to provide an optional URL for the ontology.

To distinguish between its elements and those of the included ontology, an ontology must provide a unique prefix for each ontology it extends. When the including ontology references a component from the extended ontology, it must concatenate the prefix and a period with the name of the component. For example, if the general-ontology defines the category Person, and the university-ontology ontology included it and assigned the prefix g, then the reference to Person from university-ontology would be g.Person. Prefixes may also be attached to already-prefixed references, forming a prefix chain. For example, if the university-ontology was in turn extended by the cs-dept-ontology which assigned it the prefix u, then that ontology could refer to the Person category as u.g.Person. A valid reference is one where removal of the first prefix, results in a reference that is valid in the ontology indicated by that prefix.

The base ontology is implicitly extended by all other ontologies. To refer to its categories, relations, and data types, an ontology can simply use a prefix that is a single period, as in .NUMBER. Such references are said to contain empty prefix chains. Of course, ontologies that explicitly include the base ontology are also free to use the prefixing mechanism defined above when referring to components defined in the base ontology.

Ontology extension is specified with the USE-ONTOLOGY tag, which has the following form:

<USE-ONTOLOGY ID="*ontid*"
VERSION="*ontversion*"
PREFIX="*prefix*"
[URL="*url*"]>

ID (mandatory) Specifies the extended ontology's unique identifier. This must be a valid ontology identifier, as specified above.

VERSION (mandatory) Specifies the extended ontology's version number. This must be a valid ontology version number, as specified above.

PREFIX (mandatory) Assigns a local prefix for referring to the components in the extended ontology. When the ontology refers to these components, this prefix must be appended before the component's name. It is illegal for a USE-ONTOLOGY tag to specify a prefix that is used in another USE-ONTOLOGY tag of the same ontology.

URL A URL that points to a document which contains the extended ontology. This allows agents that do not know of the ontology to locate it and incorporate it into their knowledge base.

Category Definitions

A category (or class) is a set of objects that share some common properties. Categories may be grouped as subcategories under one or more parent categories (superclasses), specifying the *is-a* relation that is commonly used in semantic networks and frame systems. The use of categories allows taxonomies to be built from the top down by subdividing known classes into smaller sets.

A category definition has the following form:

<DEF-CATEGORY NAME="$catname$"
[ISA="$pcatref_1$ $pcatref_2$ ...$pcatref_n$"]
[DESCRIPTION="$text$"]
[SHORT="$text$"]>

NAME (mandatory) The name of the defined category. This must be a letter followed by a sequence of letters, numbers, and hyphens; it may not contain any whitespace. The name is case-sensitive and must be distinct from the names of all other components defined by the ontology. It is recommended that the name be of mixed capitalization (for example, EducationalInstitution).

ISA A whitespace-delimited list of valid category references. Each of these specifies a parent category for the defined category.

DESCRIPTION A short, human-readable definition of the category.

SHORT A phrase which an agent may use to display the category to a user in a more understandable fashion than the category's name. In English ontologies, SHORT should be a singular or mass noun, lower-case unless it is a proper noun. For example, the category EducationalInstitution might have SHORT="educational institution".

Relation Definitions

A relation is a component used to describe a relationship between instances and other instances or data. A relation is composed of zero or more elements called arguments, and is equivalent to an n-ary predicate. If a relation is defined for some set of arguments, this permits SHOE documents to assert that the relation holds for certain instances of those arguments. The arguments of a relation are explicitly ordered, so each has a numbered position. Many relations are binary (have exactly two arguments).

A relation definition has the following form:

```
<DEF-RELATION NAME="relname"
              [DESCRIPTION="text"]
              [SHORT="text"]>
  arguments
</DEF-RELATION>
```

NAME (mandatory) The name of the defined relation. This must be a letter followed by a sequence of letters, numbers, and hyphens; it may not contain any whitespace. The name is case-sensitive and must be distinct from the names of all other components defined by the ontology. It is recommended that this name be of mixed capitalization with the first letter uncapitalized (for example, "isMemberOf").

DESCRIPTION A short, human-readable definition of the relation.

SHORT A phrase which an agent may use to display the relation to a user in a more understandable fashion than the relation's name. In English ontologies, SHORT should be a lower-case verb phrase for singular subjects, such that it makes some sense when appearing after the first argument but before the remaining arguments. For example, the relation "isMemberOf" might have SHORT="is a member of".

$arguments$ A sequence of two or more arguments. Each argument is defined by:

```
<DEF-ARG POS=("posint" | "FROM" | "TO")
         TYPE="datatype"
         [SHORT="text"]>
```

POS (mandatory) The position of the argument being defined. One of two formats should be followed. N-ary relations must use a positive integer to specify which argument is being defined. For the first argument, POS must equal 1 and each successive argument should be assigned the next greatest integer. It is illegal to re-use an integer or skip a number. Alternatively, binary relations (those consisting of exactly two arguments) can use the FROM and TO values to define the positions of the first and second argument, respectively. If a relation defines an argument with POS="FROM" then it must also define exactly one other argument with POS="TO". The reverse is also true.

TYPE (mandatory) The type of the argument. This must be a valid reference to a basic data type, category, or ontology-defined type. Basic data types are those defined in the base ontology, that is .STRING, .NUMBER, .DATE, and .TRUTH. These types are described in Section 4.2.2. If a category reference is provided, then the argument is considered to have an instance data type, and relation assertions may use any instance in this argument position. Ontology-defined types are described under Data Type Definitions, later in this section. The format of data stored under these types is implementation-specific.

The data type assigned to the argument determines how values used in relation assertions are interpreted. For example, if the type is .STRING, then the value 2345 will be interpreted as the string "2345", while if the value is a NUMBER, it will be interpreted as the integer 2345.

SHORT A description of each argument in the relation. This should be a lower-case singular or mass noun.

Renaming Components

Ontologies can provide aliases for other components via renaming. Any named component (i.e., category, relation, constant, type, or another rename) in the ontology or one of its ancestors can be renamed. Renaming is usually used to indicate synonyms or to reduce the length of prefix chains when referencing a component defined in a distant ancestor. For example, an ontology could rename the category cs.junk.foo.person to simply person, so long as person is not defined elsewhere in the ontology.

A renaming has the following form:

```
<DEF-RENAME FROM="compref"
            TO="newname">
```

FROM (mandatory) A reference to component being renamed. This must be a valid reference, meaning its prefix chain can be followed to locate the component in its source ontology.

TO (mandatory) The element's new name. This must be a letter followed by a sequence of letters, numbers, and hyphens, and may not contain any whitespace. The name is case-sensitive. It is considered part of the component namespace, and must be distinct from the names of all other components defined by the ontology. It is recommended that this name follow the naming conventions that apply to the renamed component.

Inference Rules

An ontology can include additional axioms by defining inference rules. An inference rule consists of a set of antecedents (one or more subclauses describing assertions that entities might make) and a set of consequents (consisting of one or more subclauses describing assertions that may be deduced if the consequents are satisfied).

The antecedents and consequents are sets of one or more subclauses, each corresponding to a logic atom. Antecedent subclauses are enclosed by <INF-IF> and </INF-IF> tags, while consequent subclauses are enclosed by <INF-THEN> and </INF-THEN> tags. An antecedent may be either a relation subclause, a category subclause, or a comparison subclause, but a consequent may only be a relation subclause or a category subclause. An inference rule with no antecedents or no consequents is invalid. The syntax of each type of subclause is described below.

The arguments in subclauses may either be constants or variables. An inference rule can have multiple variables, all of which are implicitly universally quantified. Constants require exact matches, but variables can be bound to any value that satisfies the expression. Within a particular inference rule, all variables with the same name are the same variable and must always be bound to the identical values. Variables in different rules are always considered distinct.

All variables must be limited (see Section 2.3), although a stricter definition of limited is used than that defined for datalog. Every SHOE variable must:

- appear in an antecedent that is a category or relation subclause, or
- appear with another limited variable in an antecedent that is an equal comparison subclause.

Also, each variable has a data type that can be determined from the rule.

- If the the variable appears in a category subclause, then it is of instance type.
- If the variable appears in a relation subclause, then it is of the type given by the relation's definition for that argument position. If the type is a category, then the variable is of instance type.
- If the variable appears in a comparison subclause with another typed variable, then it is assigned the type of that variable.

Note that this means that every SHOE limited variable can be assigned a data type. Any inference rule that has an unlimited variable or assigns a variable more than one data type is invalid and may be ignored. However, a variable of instance type does not have to be a member of the category which specifies the argument's type. This is because some categories subsume other and some instances are members of multiple categories.

A SHOE inference has the form:

```
<DEF-INFERENCE
                [DESCRIPTION="text"]>
  <INF-IF>
    antecedents
  </INF-IF>
  <INF-THEN>
    consequents
  </INF-THEN>
</DEF-INFERENCE>
```

DESCRIPTION A short, human-readable description for the rule.

antecedents One or more category, relation, or comparison subclauses as defined below.

consequents One or more category or relation subclauses, as defined below.

A **category subclause** is satisfied if the instance denoted by some key or variable is a member of the specified category. It has the form:

```
<CATEGORY NAME="catref"
          FOR="val"
          [[USAGE=]("VAR" | "CONST")]>
```

NAME (mandatory) A reference to the category. This must be a valid reference, meaning its prefix chain can be followed to locate the component in its source ontology.

FOR (mandatory) Specifies an instance key or a variable to be bound to an instance which has been declared to belong to this category.

USAGE Indicates if *val* refers to the key of an actual instance (CONST) or to a variable (VAR). A variable can be bound to any instance which has been declared to belong to this category. If no USAGE is specified, it defaults to CONST. Note that even when a value is specified, the USAGE= is optional.

A **relation subclause** is satisfied if the relationship holds between all of its arguments. It has the form:

<RELATION NAME="$relref$">
  $arguments$
</RELATION>

NAME (mandatory) A reference to the relation asserted for the instance. This must be a valid reference, meaning its prefix chain can be followed to locate the component's definition in its source ontology.

$arguments$ The set of arguments for the relation, specified one after another. There are two forms of relation assertions, the general form and the binary form. The general form may be used for relationships of any number of arguments. When using this form, the relation assertion must have the number of arguments specified in the relation's definition. Assertions with a different number of arguments are invalid and may be ignored.

Alternatively, the binary form can be used for relations that are defined to have exactly two arguments. In this form, POS can be FROM or TO. If the relation's definition specifies that one of the arguments is an instance type, then it may be omitted as long as the other argument is specified. In these cases, the value for the argument is assumed to be the key of the reference instance. Thus, this form allows a shorthand for many common relationships, and allows instances to be specified in more of a frame-like manner.

Regardless of form, if a relation assertion contains two or more arguments with the same POS, then it is invalid and may be ignored.

Each argument has the form:

<ARG POS=("$posint$" | "FROM" | "TO")
VALUE="val"
[[USAGE=]("VAR" | "CONST")]>

POS (mandatory) The position of the argument being defined. A positive integer indicates that this argument fits that position in the list of arguments defined for the relation. FROM is synonymous with 1. TO is synonymous with 2. Rules for use of arguments are specified below.

VALUE (mandatory) Specifies the term whose argument position is indicated by the POS attribute. For example, <ARG POS="7" VALUE="George"> declares that the constant "George" is argument 7 in the relation. If the relation's definition specifies that the type is a basic data type, then the value must be an element of that type. If the type is instance, then the value must be an instance key or a constant reference. A constant reference is prefixed with an exclamation point ('!'), as will be discussed under Constant Definitions on page 108. Finally, if the argument type is an ontology-defined type, then its syntax is implementation dependent, but SHOE agents unfamiliar with the type may treat it as a string.

The data type assigned to the argument in the relation's definition determines how the value is interpreted. For example, if the type is .STRING, then the value 2345 is interpreted as the string "2345", while if the value is a NUMBER, it is interpreted as the integer 2345.

USAGE Declares whether the element for this argument is a variable or constant, indicated by VAR and CONST, respectively. For example, <ARG POS="7" VAR VALUE="X"> declares that the variable X is argument 7 in the relation. If no value is specified, USAGE defaults to CONST. Note that even when a value is specified, the USAGE= is optional.

A **comparison subclause** is used to evaluate its arguments with respect to equality and the standard ordering operators. A comparison must have exactly two arguments. It is incorrect for an ontology to declare comparison declaration subclauses that have any other number of arguments; if this happens, the whole inference rule is incorrect and may be ignored. A comparison clause has the form:

<COMPARISON OP="*op*">
    arg_1
    arg_2
</COMPARISON>

OP (mandatory) One of the operator key words: equal, notEqual, greaterThan, greaterThanOrEqual, lessThanOrEqual, or lessThan. These all evaluate whether arg_1 is equal, not equal to, greater than, or less than arg_2, depending on the types of the arguments. The ordering of the basic data types is described in Section 4.2.2. For instance types, the values are case-sensitive and greaterThan/lessThan have no meaning.

arg_1 The first operand of the comparison. The syntax and semantics is the same as for a relation argument (see $arguments$ above).

arg_2 The second operand of the comparison. The syntax and semantics is the same as for a relation argument (see $arguments$ above).

Constant Definitions

A constant is a special instance defined in the ontology. It can be used to provide a standard name for a shared individual, such as the color red. Although red could technically be defined in an instance, and the DECLARATORS attribute of the

ONTOLOGY element could be used to specify that this instance makes standard assertions for the ontology, the key of the concept red would be a URL like http://www.cs.umd.edu/ontology-associated-instance.html#red. However, by making it a constant, the color can become an official component of the ontology and can be referred to using the ontology prefixing mechanism.

A constant is referenced by prepending a “!” and a prefix chain to its name. For example, if an ontology defined red as a constant, and some instance uses this ontology with the cs prefix, then the instance can reference red with the key !cs.red. Note that this means that there may be more than one key that references the same constant instance, depending on the particular path of prefixes chosen. All such keys should resolve to the same instance.

A constant definition has the form:

<DEF-CONSTANT NAME="$constname$"
[CATEGORY="$catref$"]>

NAME (mandatory) The name of the constant instance. This must be a letter followed by a sequence of letters, numbers, and hyphens; it must not contain any whitespace. The name is case-sensitive and must be distinct from the names of all other components defined by the ontology. It is recommended that the entire name be in lower case, so that constant names can be easily distinguished from the names of other kinds of components.

CATEGORY A reference to a single category under which the constant is to be categorized. This must be a valid reference, meaning its prefix chain can be followed to locate the component's definition in its source ontology.

Any additional assertions about the constant must be made in one or more content resources. The DECLARATORS attribute can be used to specify that these contain standard assertions of the ontology.

Data Type Definitions

Data types are sets that have specific syntactic restrictions and an implicit ordering theory associated with them. As described in Section 4.2.2, SHOE defines four basic data types: .NUMBER, .STRING, .DATE, and .TRUTH. However, certain applications may require other data types. These data types can be given names in SHOE ontologies, but SHOE does not associate any specific semantics with them. Thus, a data type definition simply allows SHOE to be used in specialized applications, where custom processors handle the specific additional data types appropriately.

Unlike the basic data types which can have empty prefix chains (that is, there is no string before the period), ontology-defined types must be referenced just as ontology-defined categories are: with a prefix chain that can be followed back to the ontology that originally defined the type.

A data type definition has the form:

```
<DEF-TYPE NAME="typename"
          [DESCRIPTION="text"]
          [SHORT="text"]>
```

NAME (mandatory) The name of the newly defined data type. This must be a letter followed by a sequence of letters, numbers, and hyphens, and may not contain any whitespace. The name is case-sensitive and must be distinct from the names of all other components defined by the ontology. It is recommended that the entire name be capitalized.

DESCRIPTION A short, human-readable definition of the data type.

SHORT A phrase which an agent may use to display the type to a user in a more understandable fashion than the data type's name. In English ontologies, SHORT should be a lower-case singular or mass noun.

4.2.4 Instance Assertions

In the previous section, we described how to define SHOE ontologies. In this section we will describe how to provide SHOE content that commits to these ontologies. An instance is a single individual or concept; it can be classified under particular categories, have properties, and be related to other instances. SHOE content is provided by a resource called a *source document*. Each source document contains one or more *reference instances*, that indicate the resource that the content is related to. A reference instance contains many assertions, and the instances referenced in these assertions (called *subject instances*) may differ from the reference instance. Usually, the reference instance is the source document, but it can also be used to specify semantic content for other resources. For example, SHOE cannot be added to a GIF image, but SHOE content in another resource can contain a reference instance that describes this image. An example reference instance is shown in Figure 4.2.

All instances must have a unique key. If the instance is a resource, then this key is typically some standard URL for the resource. If the instance is an entity described solely by that resource, then the key may be formed by adding a unique pound-suffix to the resource's URL. For example, http://www.jdoe.org/#Fido is a valid key for an instance located at http://www.jdoe.org/. It is good style for this key to correspond with an actual anchor in the document.

All SHOE content must indicate a reference instance using an <INSTANCE> tag. If the source document is an HTML document, then this tag may appear at the top level within its body (i.e., it may not be enclosed by any other tags within the BODY).

```
<INSTANCE KEY="http://univ.edu/jane/">

<!-- Use the semantics from the ontology "university-ont",
     prefixed with a "u." -->
   <USE-ONTOLOGY ID="university-ont" VERSION="1.0" PREFIX="u"
                 URL="http://www.ontology.org/univ1.0.html">

<!-- Claim some categories for this instance and others. -->
   <CATEGORY NAME="u.Chair">
   <CATEGORY NAME="u.Student" FOR="http://univ.edu/john/">

<!-- Claim some properties and relationships  -->
   <RELATION NAME="u.name">
      <ARG POS="TO" VALUE="Jane Smith">
   </RELATION>

   <RELATION NAME="u.advises">
      <ARG POS="TO" VALUE="http://univ.edu/john/">
   </RELATION>

</INSTANCE>
```

Figure 4.2: An example instance.

In an XML document, the instance element must be a subelement of the shoe element. Only SHOE tags, as described under *content* below, may appear between the open and close INSTANCE tags. The assertions made within these tags are considered knowledge relevant to the reference instance. Some documents may have multiple reference instances, each of which must have a unique key.

The syntax of a reference instance is:

<INSTANCE KEY="*key*"
    [DELEGATE-TO="del_1 del_2 ...del_n"]>
  content
</INSTANCE>

KEY (mandatory) The unique key for the instance. Keys must begin with a letter, may contain only the characters allowed in URLs, and must not contain whitespace. They are also case-sensitive. Keys that begin with an exclamation point ("!") and the key "me", in any case, are reserved.

DELEGATE-TO Specifies the URLs of resources that are permitted to make assertions on behalf of this instance. This should be a whitespace-delimited list of valid keys. Typically, the delegated resource will contain a reference instance with the same key as the permitting instance. Agents should consider all assertions made within that subinstance as if they were made by the permitting instance itself. This might be done to consolidate assertions for a web site into a single document, or to eliminate a large number of assertions from slowing down the download time of a document. If the delegated instance does not declare this special subinstance, then delegating declarative power is simply a pointer to an agent to look elsewhere for relevant SHOE knowledge.

content The content of an instance can be a combination of <USE-ONTOLOGY>, <CATEGORY>, <RELATION>, and <INSTANCE> tags. An <INSTANCE>

tag that appears within another is called a *subinstance*, and has the same syntax as other instances. The valid syntax for the other kinds of content are described below.

Committing to an Ontology

Every SHOE reference instance must commit to one or more ontologies, which provide the semantics for the knowledge about the instance. The ontologies committed to by an instance are indicated with the <USE-ONTOLOGY> tag, which has the following form:

<USE-ONTOLOGY ID="$ontid$"
VERSION="$ontversion$"
PREFIX="$prefix$"
[URL="url"]>

ID (mandatory) Specifies the extended ontology's unique identifier. This must be a valid ontology identifier, as specified in Section 4.2.3.

VERSION (mandatory) Specifies the extended ontology's version number. This must be a valid ontology version number, as specified in Section 4.2.3.

PREFIX (mandatory) Assigns a local prefix for referring to components defined in the ontology committed to. When the ontology refers to these components, this prefix must be appended before the component's name. It is illegal for a USE-ONTOLOGY tag to specify a prefix that is used in another USE-ONTOLOGY tag of the same instance or an enclosing instance.

URL A URL that points to a document which contains the ontology committed to by the instance. This allows agents that do not know of the ontology to locate it and incorporate it into their knowledge base.

Note that the syntax of the USE-ONTOLOGY tag for instances is identical to the one for ontologies.

Category Assertions

Instances may be classified, that is, they may be declared to belong to one or more categories in an ontology, using the CATEGORY tag:

<CATEGORY NAME="$catref$"
[FOR="key"]>

NAME (mandatory) A reference to the category asserted for the instance. This must be a valid reference, meaning its prefix chain can be followed to locate the category's definition in its source ontology.

FOR Contains the key of the instance which is asserted to be a member of the category. This value must be an instance key, a constant reference, or the value "me". A constant reference is prefixed with an exclamation point ('!'), as described under Constant Definitions on page 108. The value "me" is a shorthand for the key of the reference instance. If the FOR attribute does not appear, then the key is assumed to be that of the reference instance.

Relation Assertions

A reference instance may contain assertions about the properties and relationships of instances. These take the following form:

<RELATION NAME="$relref$">
$arguments$
</RELATION>

NAME (mandatory) A reference to the relation asserted for the instance. This must be a valid reference, meaning its prefix chain can be followed to locate the component's definition in its source ontology.

$arguments$ The set of arguments for the relation, specified one after another. There are two forms of relation assertions, the general form and the binary form. The general form may be used for relationships of any number of arguments. When using this form, the relation assertion must have the number of arguments specified in the relation's definition. Assertions with a different number of arguments are invalid and may be ignored.

Alternatively, the binary form can be used for relations that are defined to have exactly two arguments. In this form, POS can be FROM or TO. If the relation's definition specifies that one of the arguments is an instance type, then it may be omitted as long as the other argument is specified. In these cases, the value for the argument is assumed to be the key of the reference instance. Thus, this form allows a shorthand for many common relationships, and allows instances to be specified in more of a frame-like manner.

Regardless of form, if a relation assertion contains two or more arguments with the same POS, then it is invalid and may be ignored.

Each argument has the form:

<ARG POS=("$posint$" | "FROM" | "TO")
VALUE="val">

POS (mandatory) The position of the argument being defined. A positive integer indicates that this argument fits that position in the list of arguments defined for the relation. FROM is synonymous with 1. TO is synonymous with 2. Rules for use of arguments are specified below.

VALUE (mandatory) Specifies the term whose argument position is indicated by the POS attribute. For example, <ARG POS="7" VALUE="George"> declares that the constant "George" is argument 7 in the relation. If the relation's definition specifies that the type is a basic data type, then the value must be an element of that type. If the type is instance, then the value must be an instance key, a constant reference, or the value "me". A constant reference is prefixed with an exclamation point ('!'), as described under Constant Definitions on page 108. The value "me" is a shorthand for the key of the reference instance. In any of these cases, the specified instance is assumed to be of the category specified by the relation definition. Finally, if the argument type is an ontology-defined type, then its syntax is implementation dependent, but SHOE agents unfamiliar with the type may treat it as a string.

The data type assigned to the argument in the relation's definition determines how the value is interpreted. For example, if the type is .STRING, then the value 2345 is interpreted as the string "2345", while if the value is a NUMBER, it is interpreted as the integer 2345.

4.3 Formal Semantics

We will now present a formal semantics for the SHOE language. These semantics are based on compatible ontology perspectives, as described in Section 3.4. That definition depends on ontologies defined as tuples, a knowledge function that describes the content of resources, and a commitment function that determines which ontology a resource commits to. Thus, we need to describe how SHOE ontologies map into the required ontology structure, and how to define the knowledge and commitment functions based on the content of SHOE instances.

4.3.1 Preliminaries

Before we present the formal semantics, we must define a few useful functions and sets. First, it is useful to know which symbols of $\mathcal{L}$ correspond to what kinds of SHOE concepts. We will specify certain subsets of the symbols S that correspond to SHOE categories, relations, types, and constants. S_{cat} is the set of category symbols, S_{rel} is the set of relation symbols, S_{type} is the set of type symbols, and S_{const} is the set of constant symbols. S_{cat}, S_{rel}, and S_{type} are subsets of the predicate symbols, that is, $S_{cat} \subseteq S_P$, $S_{rel} \subseteq S_P$, and $S_{type} \subseteq S_P$. The ontology constant symbols are a subset of the language's constants symbols, that is $S_{const} \subseteq S_C$.

Additionally, we will use a set of functions to translate different aspects of the SHOE syntax to concepts in our logical model. The following functions are used:

$\mathrm{ont} : Id \times Ver \to \mathcal{O}$ Maps pairs of SHOE ontology identifiers (Id) and version numbers (Ver) to ontology structures. Since not all combinations of identifiers and version numbers have associated ontologies, this is a partial function.

$\mathrm{res} : Url \to R$ Maps a uniform resource locator (URL) from the set Url to a specific resource in R.

$\mathrm{resolve} : ShoeOnt \times CompRef \to S$ Maps a SHOE ontology from the set $ShoeOnt$ and a component reference from the set $CompRef$ to a symbol in S. This function has the following properties:

- $\mathrm{resolve}(O, p.name) = \mathrm{resolve}(O', name)$ if O' is the ontology referenced by prefix p.

- $\text{resolve}(O, p_1.p_2.\ldots.p_n.name) = \text{resolve}(O', p_2.\ldots.p_n.name)$ if O' is the ontology referenced by prefix p_1.

$\text{atom} : ShoeExp \rightarrow W$ Maps an atomic SHOE expression (an element of $ShoeExp$) to the equivalent well-formed formula in the set W. The semantics of this function are specified in Table 4.2.

$\text{var} : String \rightarrow S_X$ Maps a string to a variable symbol.

$\text{type} : S_{rel} \times integer \rightarrow S_{type} \cup S_{cat}$ Maps a relation symbol and an integer argument position to the type of the corresponding argument. In SHOE, arguments types can be data types or categories, thus $S_{type} \cup S_{cat}$.

$\text{literal} : S_{type} \times String \rightarrow S_C$ Maps a type symbol and a string to the symbol representing that string as interpreted by the type.

We will also assume that the logical language $\mathcal{L}$ contains a set of built-in predicates. These are the binary predicates $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$. For convenience, we will write them in infix form. The definitions of these predicates are as needed by the basic data types. Note, that we assume that the symbols of the data types values are distinct. This can be done without loss of generality due to simple renaming.

In the next two sections, we will provide semantics for a canonical form of SHOE. In this form, all default values are made explicit, and any invalid components are assumed to be removed.

4.3.2 Ontology Semantics

Recall that an ontology structure is a tuple $O = \langle V, A, E, R, B \rangle$, where V is a set of vocabulary symbols, A is a set of axioms, E is the set of ontologies extended by O, R

is the set of ontologies revised by O, and B is the set of ontologies with which O is backwards-compatible. In Table 4.1, we show how the tags in a SHOE ontology, identified by SO, can be used to construct an ontology structure O. In this table, we ignore attributes that have no semantics, such as SHORT and DESCRIPTION.

A SHOE <ONTOLOGY> tag indicates a new ontology O identified by an ID and VERSION. The <ONTOLOGY> tags of all ontologies determine the $\mathrm{ont}()$ function. The use of a version number implies that all ontologies with the same identifier but an earlier version number are prior versions of the ontology, and thus this information and the $\mathrm{ont}()$ function can be used to determine R, the set of ontologies revised by O. The BACKWARD-COMPATIBLE-WITH tag lists all prior version numbers with which the new ontology is backward-compatible, and can be used by the $\mathrm{ont}()$ function to identify the ontologies that form the set B. Finally, The DECLARATORS tag lists the keys of resources that contain standard assertions associated with the ontology. As will be discussed in detail in Section 4.3.3, an instance is associated with a resource r and the tags contained within the set of instances define the knowledge function K. The knowledge of declarator instances can be considered part of the axiomatization A of the ontology.

The <USE-ONTOLOGY> tag identifies an ontology extended by the current ontology, via its ID and VERSION. The set of <USE-ONTOLOGY> tags fully determines E. Note that the PREFIX attribute is only used to construct prefixed names which are used to disambiguate the vocabularies of multiple included ontologies. The $\mathrm{resolve}()$ function depends on it, but otherwise it is not used in the semantics. The URL attribute provides a location from which the ontology can be retrieved, which may be helpful to agents that consume SHOE, but has no direct bearing on the semantics.

Tags in Ontology SO	Formal Semantics
< ONTOLOGY ID="id" VERSION="ver" BACKWARD-COMPATIBLE-WITH ="$bcw_1\ bcw_2\ \ldots\ bcw_n$" DECLARATORS ="$dec_1\ dec_2\ \ldots\ dec_m$">	$\mathrm{ont}(id, ver) = O = \langle V, A, E, R, B\rangle$ $\forall v$, if $v < ver$ and $\mathrm{ont}(id, v) = O_b$, then $O_b \in R$ $\forall i, 1 \leq i \leq n$, if $\mathrm{ont}(id, bcw_i) = O_i$, then $O_i \in B$ $\forall i, 1 \leq i \leq m$, if $\mathrm{res}(dec_i) = r_i$, then $K(r_i) \subseteq A$
<USE-ONTOLOGY ID="uid" VERSION="$uver$" PREFIX="$upre$" URL="$uurl$">	if $\mathrm{ont}(uid, uver) = O_u$ then $O_u \in E$
<DEF-CATEGORY NAME="$catname$" ISA="$pcat_1\ pcat_2\ \ldots\ pcat_n$">	if $\mathrm{resolve}(SO, catname) = c$, then $c \in V, S_{cat}$ $\forall i, 1 \leq i \leq n$, if $\mathrm{resolve}(SO, pcat_i) = p$ then $[\forall x\ c(x) \rightarrow p(x)] \in A$
<DEF-RELATION NAME="$relname$" DEF-ARG POS="1" TYPE="$type_1$" DEF-ARG POS="2" TYPE="$type_2$" $\ldots$ DEF-ARG POS="n" TYPE="$type_n$" </DEF-RELATION>	if $\mathrm{resolve}(SO, relname) = p$, then $p \in V, S_{rel}$ $\forall i, 1 \leq i \leq n$, if $\mathrm{resolve}(SO, type_i) = t$ and $t \in S_{cat}$ then $[\forall x_1, \ldots, x_n\ p(x_1, x_2, \ldots, x_n)$ $\rightarrow t(x_i)] \in A$
<DEF-RENAME FROM="$oldname$" TO="$newname$">	if $s = \mathrm{resolve}(SO, oldname)$ and $s' = \mathrm{resolve}(SO, newname)$ then $s' \in V$ if $s \in S_{cat}$ or $s \in S_{type}$ then $[\forall x\ s(x) \leftrightarrow s'(x)] \in A$ if $s \in S_{rel}$ then $[\forall x_1, \ldots, x_n\ s(x_1, x_2, \ldots, x_n) \leftrightarrow$ $s'(x_1, x_2, \ldots, x_n)] \in A$ if $s \in S_{const}$ then $[s = s'] \in A$
<DEF-INFERENCE> <INF-IF> $body_1\ body_2\ ...body_n$ </INF-IF> <INF-THEN> $head_1\ head_2\ ...head_m$ </INF-THEN> </DEF-INFERENCE>	$\forall i, 1 \leq i \leq n$, $\mathrm{atom}(body_i) = IB_i$ $\forall i, 1 \leq i \leq m$, $\mathrm{atom}(head_i) = IH_i$ $[\forall(IB_1 \wedge IB_2 \wedge \cdots \wedge IB_n \rightarrow$ $IH_1 \wedge IH_2 \wedge \cdots \wedge IH_m)] \in A$
<DEF-CONSTANT NAME="$conname$" CATEGORY="$concat$">	$\mathrm{resolve}(SO, conname) = k \in S_C, S_{const}$ if $\mathrm{resolve}(SO, concat) = c$ then $[c(k)] \in A$
<DEF-TYPE NAME="$typename$">	if $\mathrm{resolve}(SO, typename) = T$ then $T \in V, S_{type}$
</ONTOLOGY>	

Table 4.1: Semantics of SHOE ontologies.

SHOE defines classes with the <DEF-CATEGORY> tag. In the semantics, classes are unary predicates. The $\mathrm{resolve}()$ function associates the name of the category with a predicate symbol in the logical language and this predicate symbol is an element of V, the ontology's vocabulary. The ISA attribute is used to specify a list of superclasses for the new class. The $\mathrm{resolve}()$ function can be used to obtain a predicate symbol from the component references to each superclass, and this is used to construct a formula that is added to the ontology's axioms A. If the symbol for the new class is c, and the symbol for the superclass is p, then this axiom is of the form $\forall x\, c(x) \rightarrow p(x)$.

A <DEF-RELATION> tag defines a new n-ary predicate, whose symbol is added to the vocabulary V of the ontology. The tag also specifies types for each argument of the relation. The semantics of these types depend on what they are. If they are one of SHOE's data types, such as .NUMBER, .DATE, .STRING, or .TRUTH, the type is only a constraint on the syntax of assertions using the relation. On the other hand, if the type is a category, then syntax cannot be used to determine if it is valid. In fact, since the Web is an open world, there is no way to definitively determine that a given instance is not a member of the specified category. Thus, in this case it is assumed that the object is of the correct type, which can be inferred by means of an additional axiom in the ontology. If the relation corresponds to an n-ary predicate p and the category for argument i corresponds to symbol t, then the axiom is of the form $\forall x_1, \ldots, x_n\, p(x_1, x_2, \ldots, x_n) \rightarrow t(x_i)$.

A <DEF-RENAME> tag provides an alias for another component used by the ontology. It can be used to establish synonyms or shorten prefix chains. If the name identifies a category, relation, or type, then a biconditional that establishes the equivalence between the symbols representing the old name and the new name can be added to the ontology's axioms A. For example, the equivalence of two n-ary

SHOE Expressions (exp)	Formal Semantics
<CATEGORY NAME="$catname$" FOR="$forkey$">	if $\mathrm{resolve}(SO, catname) = c$ and $\mathrm{resolve}(SO, forkey) = k$ then $\mathrm{atom}(exp) = c(k)$
<CATEGORY NAME="$catname$" USAGE="VAR" FOR="$forkey$">	if $\mathrm{resolve}(SO, catname) = c$ and $\mathrm{var}(forkey) = x \in S_X$ then $\mathrm{atom}(exp) = c(x)$
<RELATION NAME="$relname$"> <ARG POS="1" VALUE="val_1" <ARG POS="2" VALUE="val_2"> . . . <ARG POS="n" VALUE="val_n"> </RELATION>	$\forall i, 1 \leq i \leq n, v_i = \mathrm{resolve}(SO, val_i)$ if $\mathrm{resolve}(SO, relname) = r$ then $\mathrm{atom}(exp) = r(v_1, v_2, \ldots, v_n)$
<COMPARISON OP="op"> <ARG POS="1" VALUE="val_1"> <ARG POS="2" VALUE="val_2"> </COMPARISON>	$v_1 = \mathrm{resolve}(SO, val_1), v_2 = \mathrm{resolve}(SO, val_2)$ if op=equal then $\mathrm{atom}(exp) = [v_1 = v_2]$ if op=notEqual then $\mathrm{atom}(exp) = [v_1 \neq v_2]$ if op=greaterThan then $\mathrm{atom}(exp) = [v_1 > v_2]$ if op=greaterThanOrEqual then $\mathrm{atom}(exp) = [v_1 \geq v_2]$ if op=lessThan then $\mathrm{atom}(exp) = [v_1 < v_2]$ if op=lessThanOrEqual then $\mathrm{atom}(exp) = [v_1 \leq v_2]$
<ARG POS="i" VALUE="val_i" USAGE="VAR">	$v_i = \mathrm{var}(val_i) \in S_X$

Table 4.2: Semantics of the $\mathrm{atom}()$ function.

predicates s and s' is given by $\forall x_1, \ldots, x_n \; s(x_1, x_2, \ldots, x_n) \leftrightarrow s'(x_1, x_2, \ldots, x_n)$. If instead the name identifies a constant, then the equivalence of the instance is added.

A <DEF-INFERENCE> tag specifies an axiom to add to the theory. The axiom consists of an implication with antecedents specified by the <INF-IF> tag and consequents specified by the <INF-THEN> tag. The <INF-THEN> part can contain <RELATION> and <CATEGORY> tags, while the <INF-IF> part can contain either of those tags as well as <COMPARISON> tags. Note that by only allowing comparison subclauses to appear in the <INF-IF> part of an inference, we prevent the possibility of drawing conclusions that contradict the implicit theory of the basic data types. Each of these subclauses translates to a single atom, as specified by the $\mathrm{atom}()$ function, that is used in the axiom. The semantics of this function are given in Table

4.2. An important point is that when USAGE="VAR" is specified for a value, the resulting atom contains a variable (i.e., an element of S_X) in the appropriate place. When the axiom is formed, all variables are universally quantified.

The following restrictions apply to the axioms. First, when COMPARISON is used, both arguments must be of the same type. Second, instance types can only be used with the equal and notEqual comparisons. A variable used in a CATEGORY is always of type instance, while a variable used in a RELATION is of the type required by the argument. However, it is illegal for the same variable to be used in two arguments of different basic types (although it is legal to be used in arguments that require two different categories).

A <DEF-CONSTANT> tag specifies a constant that is identified by the ontology. The symbol associated with the constant by $\mathrm{resolve}()$ is an element of the constant symbols S_{const}. The constant can be assigned a category, and if so, the appropriate ground atom is added to the ontology's axioms A.

Finally, a <DEF-TYPE> tag is used to specify a new data type. This tag is reserved as a hook for allowing users to customize SHOE. However, SHOE does not provide means to define the syntax or semantics of this data type, any agents that use the type would need additional knowledge for recognizing the data type expressions and ordering of the values for use in <COMPARISON>. Any agent that does not recognize the type can treat it as an additional category to which no instances or subclasses can be added.

4.3.3 Instance Semantics

SHOE <INSTANCE> tags provide knowledge about resources. Recall from Chapter 3 that a resource is specified by a knowledge function K and a commitment function

Tags in Instance SI	Formal Semantics
<INSTANCE KEY="$instkey$" DELEGATE-TO="del_1 del_2 $\ldots$ del_n">	$\text{res}(instkey) = r$ $\forall i, 1 \leq i \leq n$, if $\text{res}(del_i) = d$ then $K(r) \supseteq K(d)$
<USE-ONTOLOGY ID="uid" VERSION="$uver$" PREFIX="$upre$" URL="$uurl$">	if $\text{ont}(uid, uver) = O_u$ then $C(r) = O_u$
<CATEGORY NAME="$catname$" FOR="$forkey$">	if $\text{resolve}(SI, catname) = c$ and $\text{resolve}(SI, forkey) = k$ then $[c(k)] \in K(r)$
<RELATION NAME="$relname$"> <ARG POS="1" VALUE="val_1"> <ARG POS="2" VALUE="val_2"> $\cdots$ <ARG POS="n" VALUE="val_n"> </RELATION>	Let $\text{resolve}(SI, relname) = r$ and $\forall i, 1 \leq i \leq n, t_i = \text{type}(r, i)$ if $t_i \in S_{type}$ then $v_i = \text{literal}(t_i, val_i)$ otherwise, $v_i = \text{resolve}(SI, val_i)$ then $[r(v_1, v_2, \ldots, v_n)] \in K(r)$
</INSTANCE>	

Table 4.3: Semantics of SHOE instances.

C. SHOE reference instances can be used to specify these functions. Table 4.3 summarizes the semantics for each of these tags.

The <INSTANCE> tag identifies a resource via a KEY. The $\text{res}()$ function returns the resource associated with a particular key. The rest of the tags specify the content of the knowledge and commitment functions for that resource. If the instance includes a DELEGATE-TO attribute, it specifies resources whose assertions are included by reference. Thus if an instance specifies that resource r delegates to d, the knowledge function of r should be a superset of that for d.

A SHOE instance commits to an ontology with a <USE-ONTOLOGY> tag. As with the <USE-ONTOLOGY> tag for ontologies, this tag specifies and identifier and version number for an ontology. Thus, if the resource is r and the ontology given by the $\text{ont}()$ function is O_U, then $C(r) = O_U$. Note that in SHOE, an instance can commit to many ontologies. Although the formal model does not support this directly, we can create a single virtual ontology which extends all of the ontologies committed to by the instance, and use this ontology in our commitment function.

The <CATEGORY> tag makes an assertion about the class of some instance. This assertion is a unary ground atom formed by the category predicate and a constant. This ground atom is one of the formulas returned by the knowledge function K for the resource.

The <RELATION> tag makes an assertion that is an n-ary ground atom. Assuming the RELATION element is valid, this atom is formed by applying the $\mathrm{resolve}()$ function to the relation name and to each of its argument values. The terms of the atom depend on the types specified in the relation definition. If the type is a data type, then the $\mathrm{literal}()$ function translates the value into an appropriate symbol. Otherwise the $\mathrm{resolve}()$ function is used. This ground atom is one of the formulas returned by the knowledge function K for the resource.

Chapter 5

Implementation

In this chapter, we will examine how the SHOE language can be implemented. A Semantic Web system based on SHOE requires that a number of distinct tasks be performed. We begin with an overview of these tasks and present a basic architecture for systems to support them. We then describe a number of general purpose tools that have been designed for SHOE, and show how they fit into this architecture.

5.1 Architectural Issues

A system that uses the SHOE language must take into account a number of design issues. The system must consider how ontologies are designed and possibly provide tool support for the process, it must provide tools to help users add assertions (called annotations) to their web pages, and it must decide how these assertions are accessed and then processed. In this section we will discuss each of these issues and present a general architecture.

5.1.1 Ontology Design

Before SHOE can be used, appropriate ontologies must be available. Ideally, the Semantic Web will have vast libraries of reusable ontologies, and specific domain or task ontologies can be quickly assembled by extending these ontologies. However, when there are many ontologies, an ontology author may find it difficult to locate the appropriate ontologies to reuse. An invaluable aid in this task is an index of ontologies. This index may be created by hand, as web directories are, or by a web crawler, like standard search engines. A simple index may be a set of web pages that categorize ontologies, while a more complex repository may associate a number of characteristics with each ontology so that specific searches can be issued [88]. An example of a web-based index containing over 150 ontologies is available from *http://www.daml.org/ontologies/*.

If existing ontologies do not completely fulfill the needs for a particular application, then a new ontology must be constructed. This can be a complicated and labor-intensive process, and requires the cooperation of ontology engineers and subject matter experts. A SHOE ontology is simply a text file, and as such a text editor is all that is required to create one. However, due to the complexity of ontology design, SHOE cannot have widespread success without tools that assist users in creating and editing ontologies. However, due to the focus of the SHOE project on tools that would have the highest immediate impact, a SHOE ontology editor has not been designed yet. Still, there are a number of ontology editors that can create ontologies in other languages, such as Protégé [76] and the Ontolingua Server [29], and these can provide insights into what a SHOE ontology editor might look like.

When a SHOE ontology is completed, it can be placed on the Internet so that it can be accessed by intelligent agents and SHOE-enabled search engines. It is also possible to create SHOE ontologies solely for use on intranets. These proprietary ontologies may even extend public ontologies, thus maintaining a certain level of compatibility with the rest of the Semantic Web. However, proprietary ontologies are discouraged for all except the most sensitive applications, as they hide ontology constructs that could be reused by others and thus lead to greater ontology divergence throughout the Semantic Web.

5.1.2 Annotation

One way of using SHOE is to add it directly to the web pages it describes; this process is called annotation. The first step is to choose or create an appropriate ontology, as discussed in the previous section. Then the user must select instances and describe their properties. This information is encoded in SHOE and added to web pages, but the exact method depends on the problem domain and the resources available. As with ontologies, a simple text editor is all that is required to begin annotating web pages. However, this requires familiarity with the SHOE specification and is prone to error. Therefore, we have provided the Knowledge Annotator, a graphical tool that allows users to add annotations by choosing items from lists and filling in forms. The Knowledge Annotator is described in more detail in Section 5.2.2.

However, it is tedious to use an authoring tool to generate large amounts of markup, but without plentiful markup, the Semantic Web is of limited value. In fact, detractors of the Semantic Web language approach often cite the difficulty in obtaining markup as the main reason why it will never work. Fortunately, there are many ways to generate semantic markup.

Many useful web pages have some regular structure to them, and programs (commonly called “wrappers” or “web-scrapers”) can be written to extract this data and convert it to SHOE format. Later, we will describe a tool called Running SHOE that helps users quickly create wrappers for certain kinds of pages. As XML 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 it to an ontology, and modify the scripts that produce the web pages to include the appropriate semantic markup.

Other extraction tools might include machine-learning [36, 60] or natural-language-processing techniques. NLP techniques have had success in narrow domains, and if an appropriate tool exists that works on the document collection, then it can be used to create statements that can be translated to SHOE. It should be mentioned that even if such an NLP tool is available, it is advantageous to annotate the documents with SHOE because this gives humans the opportunity to correct mistakes and allows query systems to use the information without having to reparse the text.

5.1.3 Accessing Information

Once SHOE ontologies and instances are available on the Web, SHOE agents and search engines must be able to access this information. There are two basic approaches: direct access and repository-based access. In the direct access approach, the software makes an HTTP request to the relevant web page or pages and extracts the SHOE markup. The advantage of this approach is that extracted knowledge is guaranteed to be current. However, the latency in internet connections means that this approach cannot be realistically used in situations where many pages must be searched. Therefore, it is best used to respond to specific, localized queries, where incomplete answers are expected. It may also be used to supplement ordinary browsing with additional semantic information about pages in the neighborhood of a selected page.

The repository-based access approach relies on a web-crawler to gather SHOE information and cache it in a central location, which is similar to the way contemporary search engines work. Certain constraints may be placed on such a system, such as to only visit certain hosts, only collect information regarding a particular ontology, or to answer a specific query. Queries are then issued to the repository, rather than the the Web at large. The chief advantage of the this approach is that accessing a local KB is much faster than loading web pages, and thus a complete search can be accomplished in less time. However, since a web-crawler can only process information so quickly, there is a tradeoff between coverage of the Web and freshness of the data. If the system revisits pages frequently, then there is less time for discovering new pages. Exposé, which is discussed in Section 5.2.4, is a SHOE web-crawler that enables the repository-based access approach.

5.1.4 Information Processing

Ultimately, the goal of a SHOE system is to process the data in some way. This information may be used by an intelligent web agent in the course of performing its tasks or it may be used to help a user locate useful documents. In the latter case, the system may either respond to a direct query or the user may create a standing query that the system responds to periodically with information based on its gathering efforts.

A SHOE system requires a reasoner, which is a component that can infer new facts from available information and/or answer queries from a given set of facts and rules. In a repository-based system, this is usually provided by a knowledge base system. In order to deal with large amounts of SHOE information, this system must be selected carefully. While SHOE can be implemented relatively easily in semantically sophisticated knowledge representation systems like LOOM or CYC-L, the language is intended to be feasibly implementable on top of fast, efficient knowledge representation systems with correspondingly simpler semantics.

The selection of a SHOE repository should depend on a number of factors. In order for the system to be a complete SHOE reasoner, it must have the expressivity of datalog. To handle the large volumes of data, it must use efficient secondary storage. Also, since the repository will be large, it should be leveraged by many users, and thus must support concurrent, multi-user operation. Finally, it must have a strategy for dealing with perspectives. Later in this chapter, we will discuss a number of knowledge base systems, and how they meet these criteria.

It should be noted that due to the size of the Web, complete reasoning may need to be sacrificed for the sake of improving query response times. Thus, it may be useful to have many repositories, each with different inferential capabilities and performance characteristics. Then users can select the requirements for a particular query and the repository that best fulfills these needs can provide the answer.

5.1.5 A Basic Architecture

Now that we have discussed the various requirements and choices for a SHOE system, we can propose a general architecture. The foundation of any SHOE architecture depends on the existence of web pages that contain SHOE instances, where each instance commits to one or more SHOE ontologies that are also available on the Web. A number of architectures can be built around this concept, with different sets of tools for producing and consuming this data. This idea is influenced by the design of the Web, where HTML is a lingua franca that is produced by text editors, web page editors, and databases; and processed by web browsers, search engines, and other programs.

We describe a basic architecture (shown in Figure 5.1) that was investigated extensively in this thesis. In this architecture, a number of tools can be used to create SHOE web pages. These tools include text editors, the Knowledge Annotator, Running SHOE (see Section 5.2.3), and possibly other domain specific tools. For efficiency reasons, the assertions are gathered from the web pages by a web crawler called Exposé (see Section 5.2.4), and stored in a knowledge base. The specific knowledge base system used can vary depending on the needs of the application, and multiple knowledge base systems can be used simultaneously. Finally, a number of front-ends, including SHOE Search (see Section 5.2.9), domain specific tools, or KB specific tools can be used to query the data. The generic SHOE tools will be discussed extensively in the next section.

5.2 SHOE Software

To support the implementation of SHOE we have developed a number of general purpose tools. Most of these tools are coded in Java and thus allow the development of platform independent applications and applets that can be deployed over the Web. These tools are the results of over 20,000 lines of source code, which for length reasons are not included in this thesis. However, both source and application versions

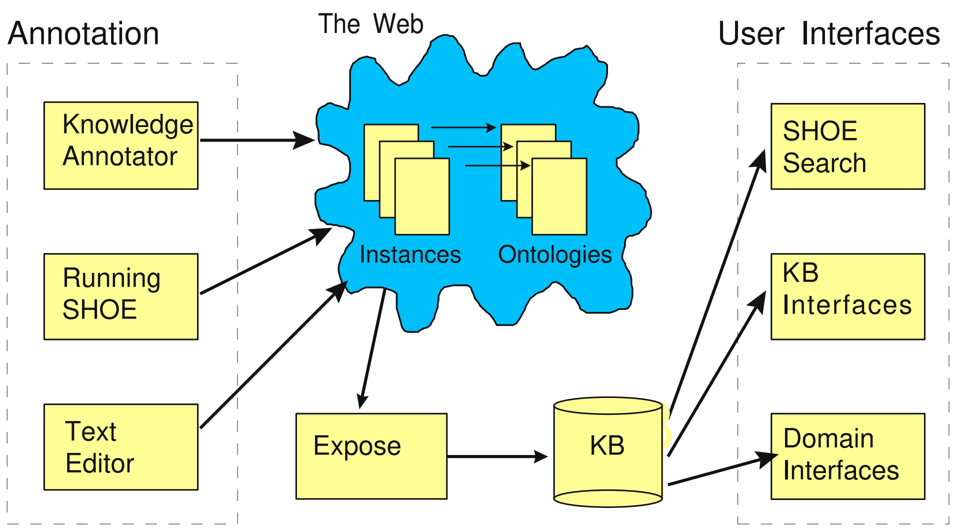

Figure 5.1: A basic SHOE architecture.

of the tools may be downloaded from

http:/www.cs.umd.edu/projects/plus/SHOE/downloads.html.

When describing the design of these tools, we will use Java's object-oriented terminology. Specifically, a *class* is a collection of data and methods that operate on that data and an *object* is instantiation of a class. The reader should be careful to distinguish between the use of these terms as programming language constructs and their use elsewhere in this paper to refer to knowledge representation concepts. Another important concept from Java is a *package*, which is a collection of Java classes meant to be used a unit.

5.2.1 The SHOE Library

The SHOE library is a Java package that can be used by other programs to parse files containing SHOE, write SHOE to files, and perform simple manipulations on various elements of the language. This library serves as a foundation for all of the Java software described later. The emphasis of the library is on KB independence, although these classes can easily be used with a KB API to store the SHOE information in a KB.

The central class, called SHOE_Doc, represents a SHOE document. This class can be used to parse a file or internet resource, or to create a new SHOE document from scratch. The document is stored in such a way that the structure and the format is preserved, while efficient access to and update of the SHOE tags within the document is still possible. SHOE_Doc has methods for returning the ontologies and instances contained within the document, and also provides methods to add and delete them.

Each SHOE tag has a corresponding class that models that element. These classes have a common ancestor and include methods for reading and interpreting the tags contained within them, modifying properties or components and validating that the object is consistent with the rules of the language. Each class uses data structures and methods that are optimized for the most common accesses to it.

SHOE ontologies are represented by an Ontology class. This class contains methods for retrieving and editing the various components of the ontology. Additionally, it has a method that will return a category's set of descendants in a tree structure, and another method that returns these structures for all of the ontology's top-level categories.

SHOE documents must be parsed in two steps. This is because SHOE is order-independent but the interpretation of some tags may depend on others in the same document. The first step ensures that the document is syntactically correct and creates the appropriate structures for each of its components. The second step ensures that the SHOE structures are internally consistent with themselves and any ontologies that they depend on.

Access to ontologies is controlled via the OntManager class. Since ontology information is used frequently, it is more efficient to access this information from memory than to access it from disk, or even worse, the Web. However, an application may require more ontologies than can be stored in memory, so the ontology manager must cache the ontologies. One of the most important features of this class is a method which resolves prefixed names. In other words, it determines precisely which ontology element is being referred to. This is non-trivial because prefix chains can result in lookups in a series of ontologies and objects can be renamed in certain ontologies. When objects that contain such prefixed names are validated, the names are resolved into an identifier that consists of the id and version of the ontology that originated the object and the name of the object within that ontology. This identifier is stored within the object to prevent unnecessary repetition of the prefix resolution process.

The ontology manager also handles ontology proxies. A proxy ontology is a copy of an ontology that is hosted at a different location. It may be needed when an Internet connection is unavailable or when connections to an ontology's original location are too slow for ordinary use. Proxy ontologies can be specified in a file that contains the identifiers, version numbers, and alternate locations for a set of ontologies. The alternate location may be a file on the local disk, or it could specify a mirror site that has better accessibility than the ontology's home site. If a proxy is specified for a particular ontology, the ontology manager will attempt to download it from from the proxy location first.

The SHOE library provides a well-tested and easy to use set of classes and methods for manipulating SHOE documents. This library can be used by any Java software for the SHOE language, and is at the core of many of the tools described in the subsequent sections.

5.2.2 Knowledge Annotator

The Knowledge Annotator is a tool that makes it easy to add SHOE knowledge to web pages by making selections and filling in forms. As can be seen in Figure 5.2, the tool has an interface that displays instances, ontologies, and assertions (referred to as claims in the figure). A variety of methods can be used to view the knowledge in the document. These include a view of the source HTML, a logical notation view, and a view that organizes assertions by subject and describes them using simple English.

The Annotator can open documents from the local disk or the Web. These are parsed using the SHOE library, and any SHOE instances contained in the document are displayed in the **Instances** panel. When an instance is selected, the ontologies it commits to and its assertions are displayed in the other panels. Instances can be added, edited and deleted. When adding an instance, the user must specify its key and an optional name. If desired, the name can be extracted from the document's TITLE with the press of a button.

Every instance must commit to at least one ontology, but may commit to more. These ontologies provide the set of categories and relations that are used to describe the instance. An ontology can be added by selecting it from a list of known ontologies, or by specifying its identifier, version number, URL, and a desired prefix. These fields can also be edited for any use-ontology, and an ontology can be deleted

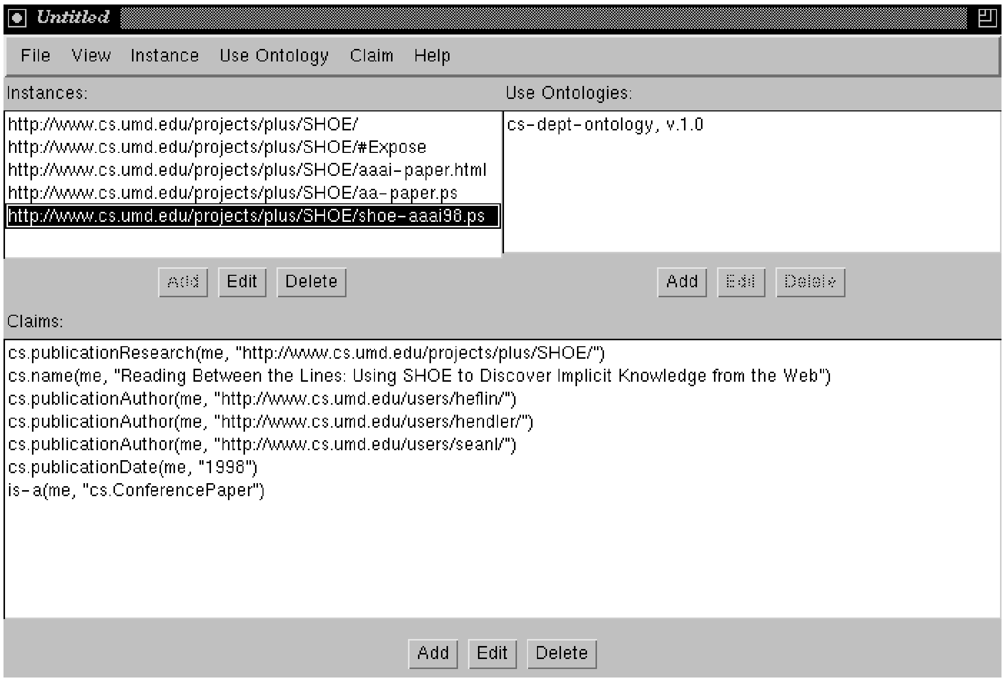

Figure 5.2: The Knowledge Annotator.

from the list of ontologies committed to by an instance. When the user selects an ontology, the OntManager class of the SHOE library is used to retrieve it, so that it may be used in the next step.

After the user has selected an instance and committed to an ontology, he can add, edit, and delete assertions. When adding or editing an assertion, the user is presented with a window like the one shown in Figure 5.3. This window contains a list from which the user can select the ontology that contains the desired category or relation. This will place a value in the **Prefix** field and cause the set of elements from the ontology to appear in the adjacent window. This list can be filtered to show just the relations or just the categories using the **Filter** choice list. A relation or category assertion is added by selecting the type of assertion from the **Type** choice list, and then selecting a name from the **Elements** list. If a relation is selected, the types of its

Figure 5.3: The Knowledge Annotator's assertion window.

arguments are displayed next to the appropriate fields, and the user can enter values for these fields. When the user enters an instance key in one of these fields, the available relations will automatically be filtered to display only those relations where the instance is of the correct type for that argument. This can help the user focus in on the relevant relations of large ontologies. If a category is selected, the user must supply the key of the instance. To reduce the amount of work for the user, the first argument of a new relation assertion or the key of a new category assertion defaults to the subject of the assertion selected in the main window. The addition or edit of the assertion can be confirmed with the **OK** button, which closes the window, or the **New** button, which allows another assertion to be added.

The Annotator automatically checks that all supplied values are valid and adds the correct SHOE tags to the document. New tags are inserted near the end of the document's body or within the correct instance. The document can then be saved to a file.

The Knowledge Annotator can be used by inexperienced users to create simple SHOE markup for their pages. By guiding them through the process and prompting them with forms, it can create valid SHOE without the user having to know the underlying syntax. For these reasons, only a rudimentary understanding of SHOE is necessary to markup web pages. However, for large markup efforts, the Annotator can be painstakingly slow. In the next section, we describe Running SHOE, which can help in many situations.

5.2.3 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. Running SHOE (see Figure 5.4) is a tool that helps users specify how to extract SHOE markup from these kinds of web pages. The user selects a page to mark up 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.

For each field the user identifies, he must indicate whether its value is a literal or a URL. Many web pages use relative URLs, which only specify a path relative to the URL of the current page. When the user indicates that a field contains URLs, Running SHOE knows to expand the relative URLs by using the page's URL as a base. This is useful because in general URLs make good keys for SHOE instances,

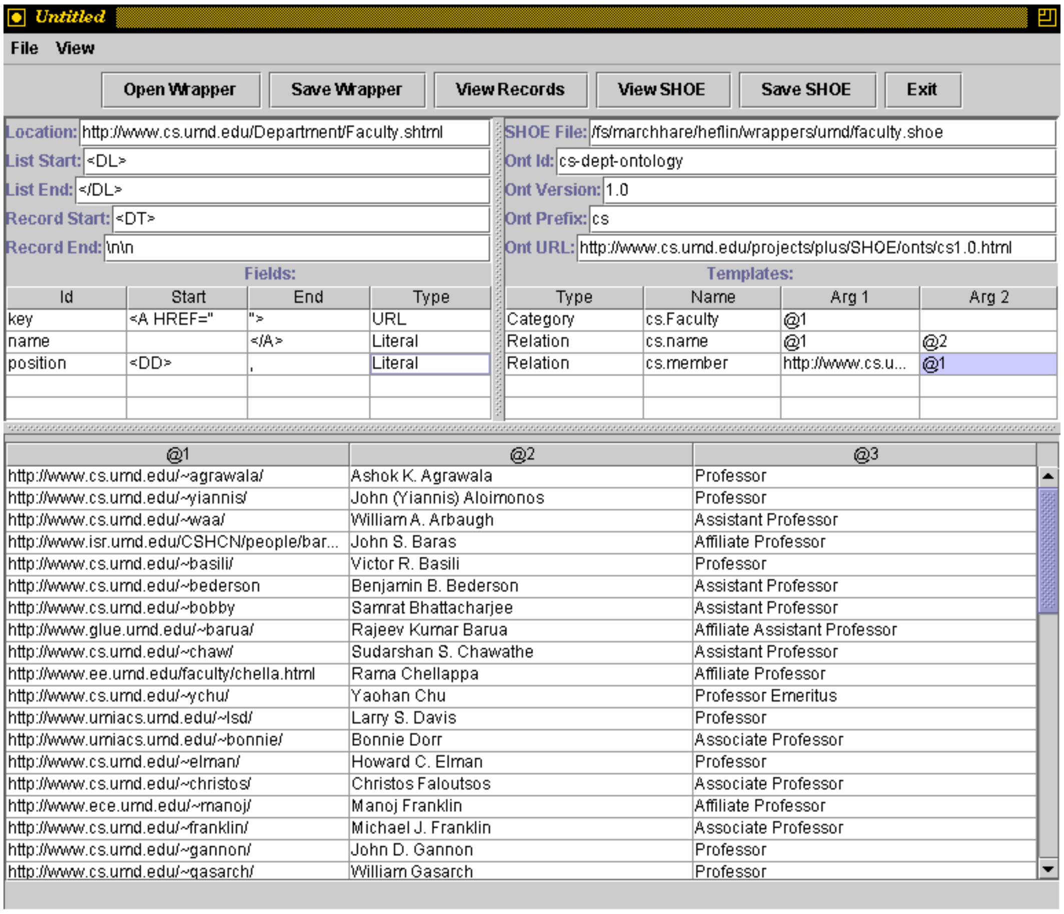

Figure 5.4: Running SHOE.

but relative URLs are poor keys because they do not uniquely identify a web page. If we expand the relative URLs, then they become unique, and we can use them as keys.

After the user has specified the delimiters and pressed the **View Records** button, the tool displays a table with a row for each record and a column for each field. Running SHOE creates this table by scanning through the document, and extracting records and fields based on the specified delimiters. Since, irregularities in a page's HTML code can cause the program to extract fields or records improperly, this table is used to verify the results and perform corrections 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 category and relation assertions. The ontology information, including identifier, version number, prefix, and URL, are used to create a USE-ONTOLOGY tag. The templates are used to specify the type of assertion, its name, and its arguments. For category assertions, the first argument is the key of the instance being classified, while the second argument is not used. For relation assertions, both arguments must be specified. Either a literal or a field reference can be used as a template argument. A literal is simply a string that will appear in SHOE tags as is. A field reference is of the form @i and references the ith field, where fields are numbered based on the order of the table's columns.

By pressing the **View SHOE** button, the user instructs the tool to extract and display the set of SHOE tags. Essentially, the tool takes each assertion template and iterates through the table creating a tag for each record, where field references are replaced by the record's value for the identified column. If the user is satisfied with the results, then he can save them to a file with the **Save SHOE** button.

The pages that work best with Running SHOE tend to have long lists or tables of things, where each item or row contains a hyperlink to some concept's homepage. Using this tool, a trained user can extract substantial markup from these web pages in minutes. Furthermore, Running SHOE lets users save and retrieve templates, so it is easy to regenerate new SHOE markup if the page's content changes. Although Running SHOE was originally designed to extract SHOE from HTML pages, it can also be used with XML document. In fact, since XML more clearly delineates the content of the document, it is even easier to use Running SHOE with XML documents than with HTML ones.

5.2.4 Exposé

After SHOE content has been created, whether by the Knowledge Annotator, Running SHOE, or other tools, it can be accessed by Exposé, a web-crawler that searches for web pages with SHOE markup. Exposé stores the knowledge it gathers in a knowledge base, and thus can be used as part of a repository-based system. The web-crawler is initialized by specifying a starting URL, a repository, and a set of constraints on which web sites or directories it may visit. These constraints allow the search to focus on sources of information that are known to be of high quality and can be used to keep the agent from accumulating more information than the knowledge base can handle. Exposé can either build a new repository of SHOE information or revisit a set of web pages to refresh an existing repository.

A web-crawler essentially performs a graph traversal where the nodes are web pages and the arcs are the hypertext links between them. Exposé maintains an open list of URLs to visit, and a closed list of URLs that have already been visited. When visiting web pages, it follows standard web robot etiquette by not requesting pages that have been disallowed by a server's robot.txt file and by waiting 30 seconds between page requests, so as not to overload a server.

The progress of the crawl can be monitored and paused by the user. Exposé's display shows each URL that has been requested, the timestamp of the request, and the status of the request. The tool also keeps track of the total number of page requests, how many of the pages have SHOE on them, and how many requests resulted in errors. At any time, the user can pause the search, and then resume it later.

Upon discovering a new URL, Exposé assigns it a cost and uses this cost to determine where it will be placed in a queue of URLs to be visited. In this way, the cost function determines the order of the traversal. We assume that SHOE pages will tend to be localized and interconnected. For this reason, we currently use a cost function which increases with distance from the start node, where paths through non-SHOE 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.

When Exposé loads a web page, it parses it using the SHOE library, identifies all of the hypertext links, category instances, and relation arguments within the page, and evaluates each new URL as above. Finally, the agent uses the SHOE KB Library API to store SHOE category and relation assertion in a specified knowledge base. This API, described in the next section, makes it easy to adapt Exposé for use with different knowledge bases.

5.2.5 The SHOE KB Library

The SHOE KB library is a Java package that provides a generic API for storing SHOE data and accessing a query engine. Applications that use this API can be easily modified to use a different reasoning system, thus allowing them to execute in a different portion of the completeness / execution time tradeoff space.

ShoeKb, the main class of the SHOE KB Library API contains methods for storing ontologies, storing SHOE document data, and issuing queries to a repository. It maintains a catalog of all ontologies stored in the KB, and provides a renaming method that distinguishes between components defined in different ontologies. When storing document data, it deletes any assertions that were in a previous version of the document, since they are no longer reflected on the Web. The class also allows a default ontology to be specified, which is used to disambiguate query predicates and identify the basis of the query's perspective.

The ShoeKb class also provides the option to forward-chain a special kind of inference. SHOE's formal semantics state that if an instance appears in an argument of a relation which is of an instance type, then a logical consequence of the assertion is that the instance is a member of the required category. However, this can result in the addition of a large number of rules to the KB. Therefore, the library supports the option to forward-chain these particular consequences and explicitly store them in the knowledge base.

Logical sentences are represented by a Sentence class. Subtypes of this class include Query, which is a conjunctive query, and Axiom which is a Horn clause. Both queries and axioms are composed of Atom objects.

The ShoeKb class uses a KBInterface which defines methods to access the basic primitives of knowledge bases. KBInterface is loosely based on a restricted version of OKBC [17], and contains methods for connecting to knowledge bases, storing sentences in them, and issuing queries. This class can be subclassed with implementations for specific knowledge representation systems, and thus provides a generic API for standard knowledge base functionality.

The SHOE KB library is used by all of the SHOE tools that must access a knowledge base. By using a standard API, these applications can be easily modified to use different backends. All that is needed is a simple subclass of ShoeKb or KBInterface that interfaces with the knowledge base. Subclasses of the ShoeKb class have been implemented for XSB, Parka, and OKBC-compliant knowledge bases. These systems are discussed in the following sections.

5.2.6 XSB

XSB [83] is an open source, logic programming system that can be used as a deductive database engine. It is more expressive than datalog, and thus can be used to completely implement SHOE. XSB's syntax is similar to that of Prolog, but while Prolog will not terminate for some datalog programs, XSB uses tabling to ensure that all datalog programs terminate.

We will now describe how compatible ontology perspectives can be implemented in XSB. In this approach, the SHOE ontologies and instances are translated into an XSB program that can evaluate contextualized queries. Although this discussion is specific to XSB, with minor modifications the approach can be applied to other logic programming systems.

First, we must consider a naming convention for predicates in the deductive database. Since two ontologies may use the same term with different meanings, the definitions of such terms could interact in unintended ways, and thus a renaming must be applied. A simple convention is to assign a sequence number to each ontology and append this number to each term defined by the ontology. The sequence numbers can be stored in the deductive database using a ontSeq/2 predicate which associates an ontology with its sequence number.

As shown in the formal semantics (see Section 4.3), SHOE ontologies can be translated into five-tuple ontology structures $\langle V, A, E, R, B\rangle$. The axioms A of these structures can be written as Horn clauses, which with some modification can be used in the XSB program. First, a naming substitution as describe above must be applied. Second, we must localize the clauses so that they do not fire in inappropriate contexts. A naive approach would simply generate a new program for each query, thus ensuring that only the appropriate rules are included, but the size of the Semantic Web would make this extremely expensive. Instead, it is more efficient to create a single program that includes some sort of switch to indicate the context. There are two possible ways of doing this: one is to add a context argument to each predicate and set this argument to the ontology identifier; the other is to create a context predicate and add this structure to the body of each rule. For example, if the O_{univ} ontology was assigned the sequence number 1, and contained the axiom $\forall x, Organization(x) \leftarrow University(x)$, then the first method would translate the axiom into:

```
organization1(univ,X) :- university1(univ,X).
```

Alternatively, the second method would translate the axiom as:

```
organization1(X) :- context(univ), university1(X).
```

We find this second method more natural, and choose it over the first one. To ensure that the program fails gracefully, a default fail rule is created for each ontology term that appears only in the body of a rule.

When an ontology O_1 extends another ontology O_2, that is $O_2 \in E_1$, then any perspective based on O_1 must also include all axioms of O_2. Another way of thinking about this is that the context of O_1 implies the O_2 context. Thus, if we had a univ ontology that extended a gen ontology, our logic program would need the rule:

```
context(gen) :- context(univ).
```

This says that whenever the context is univ, the context is also gen. However, as we will explain later, context is a dynamic predicate, which must be defined using the assert predicate in XSB. Thus, the XSB program actually has:

```
:- assert((context(gen) :- context(univ))).
```

We also need rules to indicate how a backward-compatible ontology can use data that committed to a previous version. When the revision is the current context, we want all facts that use predicates from the earlier ontology to imply facts using predicates defined in the newer ontology. For example, assume that the O_{cs2} ontology declares itself to be backward-compatible with the O_{cs1} ontology (that is $O_{cs1} \in B_{cs2}$). If the sequence numbers of the O_{cs1} and O_{cs2} ontologies are 1 and 2, respectively, the *Person* term can be made compatible across ontologies by adding the following rule to our program:

```
person2(X) :- context(cs2), person1(X).
```

This says that if the context is cs2 and X is a person1, then X is also a person2. In other words, from the perspective of O_{cs2}, a O_{cs1}*:Person* is also a O_{cs2}*:Person*.

In the formal semantics, DEF-RENAME is equivalent to a biconditional axiom of the form $s(x_1, x_2, \ldots, x_n) \leftrightarrow s'(x_1, x_2, \ldots, x_n)$. This is easily implemented in the program by splitting it into two Horn clauses. Note, that although it may be tempting to perform pre-processing to simply use the same predicate wherever the name or its alias appears, this would be incorrect. For example, assume we have ontologies A, B, and B', where B extends A and B' is backward-compatible with B, but does not extend A. Also assume that $B : q$ renames $A : p$ and B' defines q differently than B does. If we used the renaming approach, then our compatibility axiom would be

$A : p \rightarrow B' : q$, but this is false because resources that commit to A are *not* in the perspective based on B'! Instead, we must be able to say $B : q \rightarrow B' : q$.

Now we can turn our attention to SHOE instances. According to the formal semantics, assertions are simply ground atoms, and the corresponding atoms for a resource can be retrieved using the knowledge function K. It is easy to translate these atoms into facts in the deductive database. For relation assertions, we simply need to find the correct predicate for any given relation name, while for category assertions, we must convert the category to the appropriate unary predicate. In both cases, the commitment function C specifies the ontology that the resource commits to, and thus determines how to rename the predicates.

Finally, we add a helper predicate for issuing queries to the knowledge base using a particular context. Since all of the rules in the program use the context predicate to determine if they should fire, a query must be able to create a fact for this predicate. We can use assert to dynamically update context before we issue our query, and then retract the context after the query has completed. In order to return the entire set of answers to a query, instead of a tuple at a time, the query predicate must call the Prolog predicate setof between the assert and retract. Thus, our query predicate looks like:

```
query(C, V, Q, L) :- assert(context(C)),
                     setof(V, Q, L),
                     retract(context(C)).
```

The query predicate takes a context C, a variable ordering V, and a query Q as inputs, and returns a list variable L that binds to the answers.

Given a set of SHOE documents, the SHOE KB API for XSB constructs an XSB program using this method. This program can then be compiled for maximum performance. The API also has the ability to start an XSB process, load a program, send queries, and parse the answers.

Although XSB is capable of implementing SHOE's full semantics, this approach does have its limitations. First, programs cannot be modified dynamically, and to handle new ontologies or instances, a new program must be created and compiled. It may be possible to alleviate this by using XSB's feature for storing the extensional database in a relational database. If this was done, then assertions could easily be added and deleted using the database's standard features. Another problem is that XSB is a single-user system, which means that a new XSB process must be started for each user. Since the knowledge base will be very large, this can be extremely inefficient. A potential solution is to create a client-server interface to a single XSB process, but this would still be unable to process queries concurrently. A final problem with XSB is that it may not scale to the sizes needed for semantic web knowledge bases. Future work will address these problems.

5.2.7 Parka

Parka [27, 84] is a high-performance knowledge representation system whose roots lie in semantic networks and frame systems. It is capable of performing complex queries over very large knowledge bases in less than a second [84]. For example, when used with a Unified Medical Language System (UMLS) knowledge base consisting of almost 2 million assertions, Parka was able to execute complex recognition queries in under 2 seconds. One of Parka's strong suits is that it can work on top of relational database systems, taking advantage of their transaction guarantees while still performing very fast queries.

Parka represents the world with categories, instances, and n-ary predicates and can infer category membership and inheritance. It includes a built-in subcategory relation between categories (called *isa*) and a categorization relation between an instance and a category (called *instanceof*). It also includes a predicate for partial string matching, and a number of comparison predicates.

Parka can support SHOE's most widely-used semantics directly. As with XSB, a renaming must be applied to guarantee the uniqueness of terms defined in different ontologies. The standard category axioms can be represented by Parka's *isa* relation, the membership of an instance in a category can be represented by the *instanceof* predicate, and all relation definitions can be accommodated by defining new predicates. Additionally, the Parka version of the SHOE KB API automatically computes and explicitly stores the inferred types for certain relations. Furthermore, renaming is handled by pre-processing step, which computes the source ontology and component for any name. A simple name substitution is then used to ensure that the correct predicate is selected. Thus, the only language component that Parka does not support is a general inference rule mechanism.

A Parka knowledge base can be updated dynamically, which is advantageous for a system that must mirror the rapidly changing Web. In order to provide the best possible snapshot of the Web, the knowledge base must delete assertions that no longer appear in a resource. To enable Parka to delete these assertions, we have to keep track of each assertion's source. One solution would be to represent source information with an extra argument to each predicate, but the *isa* and *instanceof* links are built-in binary predicates in Parka. Thus, this approach could not be used without changing the internal workings of the knowledge base. An alternative is to store two facts for each assertion. The first fact ignores the source, and can be used normally in Parka. The second fact uses a *claims* predicate to link the source to the first fact. Although this results in twice as many assertions being made to the knowledge base, it preserves inheritance while keeping queries straightforward. The *claims* predicate can be used to find the the source of an assertion or to find all assertions from a particular source.

Parka uses sockets for multi-user, client-server operation. When the ParkaInterface class (a subclass of KBInterface) is instantiated, it sends a message to the Parka listener, which responds by creating a new process and establishing a socket for communication with the client. When the issueQuery method is called, the query is translated into Parka format and sent to the server via the socket. After the server processes the query, the client uses the socket to retrieve the answers.

Parka's chief advantages are that it allows dynamic updates and that it can be run in a multi-user, client-server mode. However, Parka does not provide the ability to partition its knowledge base, and because the inference across *isa* and *instanceof* links is built-in, true partitioning cannot be accomplished within the system. Furthermore, because Parka does not have arbitrary axioms, it does not support the SHOE notion of ontology backward-compatibility. Thus a Parka knowledge base is best used to represent a single extended ontology perspective, preferably based on many ontologies.

5.2.8 Relational Database Management Systems

It is also possible to use a relational database management system (RDBMS) as a SHOE repository. RDBMSs have been designed to efficiently answer queries over large databases, and thus scale better than XSB or Parka. However, this efficiency comes at a cost: there is no way to explicitly specify inference.

We will now sketch a method for representing SHOE in an RDBMS. First, as with XSB and Parka, a renaming must be applied to distinguish between predicates in different ontologies. Then each n-ary SHOE relation is represented by a database relation with n attributes, while each category is represented by a unary relation. Every SHOE relation assertion and category assertion is a tuple in one of the database relations.

Even certain types of inference rules can be implemented in RDBMSs. As described by Ullman [86, Chapter 3], for any set of safe, non-recursive datalog rules with stratified negation, there exists an expression in relational algebra that computes a relation for each predicate in the set of rules. Thus database views could be used to compute each predicate. Although the semantics of SHOE are safe and include no negation, SHOE rules can be recursive. Therefore, some but not all, of the rules could be implemented using views. Depending on the RDBMS, some recursive predicates may even be computable. For example, some commercial RDBMSs include operators to compute the transitive closure of a relation (e.g., the CONNECT WITH option in the Oracle SELECT operator).

A SHOE repository that uses a RDBMS sacrifices completeness for improved performance, giving us another option for applications. Although we have not yet implemented a version of the SHOE KB library for a RDBMS, this will be accomplished in future work.

5.2.9 SHOE Search

SHOE Search [47] is a tool used to query the information that has been loaded into a SHOE KB. It gives users a new way to browse the web by allowing them to submit structured queries and open documents based on the results. The basic idea is that if queries are issued within a context, the tool can prompt the user with context specific information and can more accurately locate the information desired by the user. A

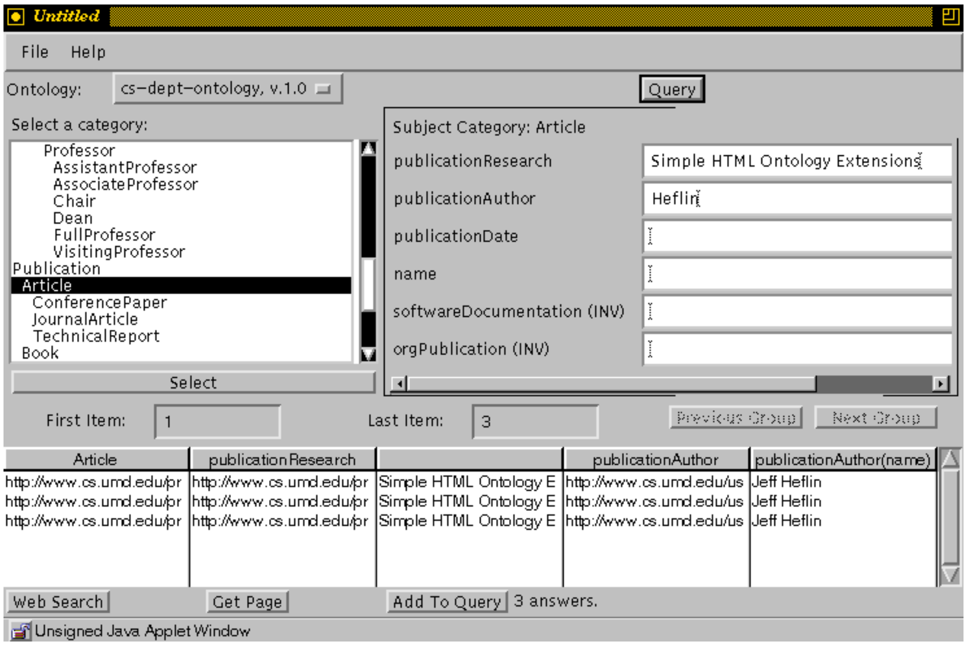

Figure 5.5: SHOE Search.

screen shot of SHOE Search is shown in Figure 5.5.

The user selects a context by choosing an ontology from a drop-down list. The list of available ontologies are those that are known by the underlying KB. The identifiers and the version numbers of each ontology are displayed, so that the user may choose to issue queries against earlier versions of ontologies.

After the user chooses an ontology, the system populates a list of categories that are defined in that ontology. This list is organized so that the specializations of each category are indented beneath it. This taxonomy makes it possible for the user to quickly determine the kinds of objects that are described in the ontology and to choose a class that is of sufficient granularity for his or her needs. Since one of the main purposes of choosing an ontology is to reduce the number of choices that the user will have to make subsequently, the list of categories generally does not include categories defined in ontologies extended by the selected ontology, even if they are ancestors of categories defined locally. It is assumed that if these categories are of interest to the user, then he can first select an appropriate ontology. However, ontologies may rename an element from an extended ontology and effectively “import” them. Such categories are included in the list, and displayed with their local name.

When the user chooses a category and presses the **Select** button, the system responds with a set of properties that are applicable for that category (in a frame system, this would essentially be the slots of the selected frame). Applicable properties are inheritable; thus any properties that apply to an ancestor of the selected category are also included in the set. However, as with the list of available categories, it is important to provide some filtering for the user, so only those relations that are defined or aliased in the selected ontology will appear, even if other ontologies define relations that could be relevant.

Technically, SHOE’s properties are relations which can have some number of typed arguments. As such, a property of a class can be considered a relation where the first argument must be a member of that class. However, this makes the determination of properties dependent on the somewhat arbitrary ordering of arguments as chosen by the ontology designer. That is, the relation *worksFor(Person, Organization)* would be a property of the class *Person*, but the inverse relation *hasEmployee(Organization, Person)* would be a property of *Organization*. In order to prevent SHOE Search queries from be restricted by these kinds of representational decisions, a relation in which the class is a subclass of the second argument is considered an inverse property and is included in the set available to the user. Such properties are clearly labeled in the display.

The property list allows the user to issue a query by example. He can type in values for one or more of the properties, and the system will only return those instances that match all of the specified values. Some of these property values are literals (i.e., strings, numbers, etc.) while others may be instances of classes. In the latter case, the user may not know the keys for these instances, since they are typically long URLs. Therefore, arbitrary strings are allowed in these fields and the query will attempt to match these strings to the names of instances. To increase the chance of a match, case-insensitivity and partial string matching are used.

When the user presses the **Query** button, the system constructs a conjunctive query and uses the SHOE KB library to issue it to a KB. The first atom of the query specifies that the instance must be of the selected category, e.g., $instanceOf(x, Person)$. The remaining atoms depend on the type of the argument that the value represents. In the case of numbers, the atom is simply looking for an instance that has the specified value for the relation. In the case of strings, two atoms are added, one to find the values of the relation, and the other to perform a partial string match on them to the string specified by the user. The latter of these atoms assumes the presence of a $stringMatch$ predicate which is true if the first argument's string contains the second argument's string. Finally, if the type of the argument is a category, then three clauses are added: one to get the values for the relation, one to get the corresponding names of these instance keys, and a third to partially match the name strings to the string specified by the user. Note that even if the user only specified values for two properties, the resulting query could contain as many as seven conjuncts. One of the advantages of SHOE Search is that useful but complex queries are constructed automatically. For example, the query constructed by the user in Figure 5.5 corresponds to a Parka query of the form:

$instanceOf(Article, x_1) \wedge publicationResearch(x_1, x_2) \wedge name(x_2, n_2) \wedge$
$stringMatch(n_2,$ "Simple HTML Ontology Extensions" $) \wedge$
$publicationAuthor(x_1, x_3) \wedge name(x_3, n_3) \wedge stringMatch(n_3,$ "Heflin" $)$

Many users would have difficulty constructing such queries by hand.

When the KB returns the results of the query, they are displayed in tabular format. The KB is likely to return many duplicate results; some of these will be due to redundancies of different web pages, others might be because the same page was visited many times using different URLs. Either way, duplicate results would simply clutter the display, and therefore they are removed before the system displays them. Generally, both the names and keys are displayed for related instances, so that the user can distinguish between instances that happen to have identical names. If the user clicks on an instance key, whether it is the instance that matches the query, or one that matches one of its properties, the corresponding web page is opened in a new browser window. This allows the user to browse the Web with more control over the queries.

Sometimes users may have trouble deciding what values to use for a given property and may end up getting no results because incorrect values were entered. To remedy this problem, we have added a **Find** button next to each property that finds valid values for that property. If this button is pressed the system will essentially issue a query to find all instances that have a value for the selected property and return those values in the tabular display. The user may then select one of these values and press the **Add To Query** button to insert it into the query field for the property. In order to do this, the system always keeps track of which columns of query results correspond to which properties.

The user may wish to view the values for a certain property without restricting the query. The **Show** checkbox allows the user to specify that an additional property should be displayed in the results. If the user specifies a value for a property, then its **Show** box is checked by default. If the box is checked but no property value was specified, then an extra atom is added to the query, where its predicate is determined by the property and a new variable is used for its value. Since the current system only supports conjunctive queries, this option can have unintuitive results. For example, if the user chooses to show a property for which no instances have a value, then no answers are returned, even if there are many possible answers for the rest of the query.

The **Show** checkbox and the **Add To Query** button can be used together to help the user gradually filter results and find the desired instances. The user starts by checking some of the **Show** boxes and issuing a query. One of the results can be selected and added to the query. When the query is reissued, fewer results should be returned. By repeating this process the user can continue to reduce the results returned to a manageable set.

It may be the case that not all of the relevant web pages are described by SHOE markup. In such cases, the standard query method of SHOE Search will not be able to return an answer, or may only return partial answers. Therefore, we have a **Web Search** feature that will translate the user's query into a similar search engine query and allow him to submit it to any one of a number of popular search engines. Using SHOE Search in this way has two advantages over using the search engines directly. First, by prompting the user for values of properties, it increases the chance that the user will provide distinguishing information for the desired results. Second, by automatically creating the query, SHOE Search can take advantage of helpful features that are often overlooked by users such as quoting phrases or using the plus sign to indicate a mandatory term. The quality of results for these queries vary depending on the type of query and the search engine used.

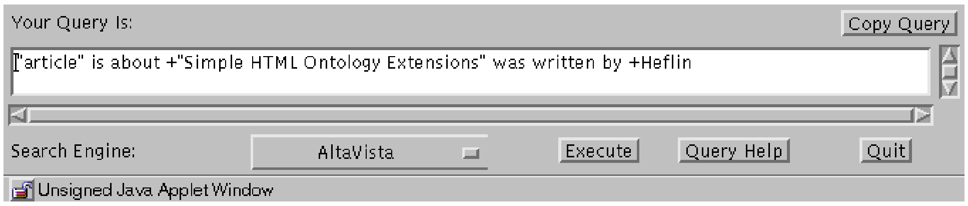

Figure 5.6: SHOE Search's web search window.

Figure 5.6 displays a window with the search engine query that is generated from the inputs specified in Figure 5.5. Currently, we build a query string that consists of a quoted short name for the selected category and, for each property value specified by the user, a short phrase describing the property followed by the user's value, which is quoted and preceded by a plus sign. For search engines with advanced query capabilities, these queries could be expanded to include synonyms for terms using disjunction or positional information could be used to relate properties to their values.

SHOE Search provides a simple interface for querying the Semantic Web. It is primarily used as a Java applet, and as such is executed on the machine of each user who opens it. Since it uses the SHOE KB library, it can be easily tailored for use with a range of knowledge representation systems.

5.3 Summary

In this section we have described the issues of implementing a SHOE system and presented a basic architecture. We have developed numerous tools to help in all aspects of the process, including creating SHOE assertions, discovering them with a web-crawler, storing them in a knowledge base, and using them to answer queries and retrieve documents. In the next section we examine the practical issues of the language by using the tools to solve problems in two different domains.

Chapter 6

Case Studies

In this chapter, we will discuss the application of SHOE to two different domains. The first case study describes the use of SHOE in the domain of computer science departments. In this study, we attempted to quickly generate a lot of SHOE from existing web pages. Another key feature of the study is an examination of a non-trivial ontology revision. The computer science case study was essentially an *in vitro* experiment, because only members of the research team participated in it, but the second case study, which considered the domain of food safety, was more *in vivo*. It attempted to solve a real-world problem and was developed as a cooperation between the SHOE team and members of the Joint Institute for Food Safety and Nutrition (JIFSAN). A key feature of this case study is the use of SHOE with a domain-specific tool. In both case studies, we discuss the processes of designing the ontologies, adding SHOE markup to web pages, and providing useful services to users.

6.1 Computer Science Departments

The computer science department application was intended to be a proof of concept for the SHOE language. We chose this domain because it was well-defined and familiar to the research team. Our objective was to evaluate the ease of adding SHOE to web pages, the types of queries that could be constructed, and the performance of SHOE queries in a simple environment. In this section we describe the development of the ontology and different methods for annotating web pages, particularly focusing on the problem of rapid creation of SHOE assertions. Finally, we provide a detailed example of how the need for ontology revision can arise and show how SHOE's backward-compatibility feature can be used in solving the problem. Some of the material in this section has appeared in a previous article [49].

6.1.1 The Computer Science Department Ontology

The first step in developing any SHOE application is to create an ontology. We named this domain's ontology cs-dept-ontology and assigned the version number 1.0 to it. In SHOE this is specified by the following tag:

```
<ONTOLOGY ID="cs-dept-ontology" VERSION="1.0">
```

Normally, the ontology should extend other ontologies, but since this was the first SHOE ontology, we could only extend the base ontology. This is specified as:

```
<USE-ONTOLOGY ID="base-ontology" VERSION="1.0" PREFIX="base"
   URL="http://www.cs.umd.edu/projects/plus/SHOE/onts/base1.0.html">
```

The next step was to identify the top-level categories of the domain. These categories should be broad classifications of the types of objects found in the domain. For the CS domain, the top-level categories were Person, Organization, Publication and Work. All of these categories were subcategories of SHOEEntity from the base ontology, which is SHOE's most generic category. Next, we refined the categories by identifying more detailed subcategories. For example, subcategories of Person included Faculty and Student, while subcategories of Organization included

Department and University. Many of the subcategories also had subcategories of their own. For example, subcategories of Professor were AssistantProfessor, AssociateProfessor, and FullProfessor. In SHOE, each category is defined by a DEF-CATEGORY tag, as shown below:

```
<DEF-CATEGORY NAME="Student" ISA="Person">
```

In some cases, a category had more than one supercategory. For example, the supercategories of Chair are AdministrativeStaff and Professor. In SHOE, this is defined by listing multiple category names in the value of the ISA attribute, where they are separated by whitespace as shown here:

```
<DEF-CATEGORY NAME="Chair"
              ISA="AdministrativeStaff Professor">
```

After identifying the categories and organizing them in a taxonomy, we needed to define properties and relationships for these categories. An Organization might be described by its name, its parent organization, and its members. A publication might be described by its name, authors, and publication date. In SHOE, these properties are defined with the DEF-RELATION element. We created relations called subOrganization, member, publicationAuthor, and publicationDate. For each relation, its arguments must be identified. For example, subOrganization is a relationship between two organizations, and thus it should have two arguments, both of type Organization. This is written in SHOE as:

```
<DEF-RELATION NAME="subOrganization" SHORT="is part of">
   <DEF-ARG POS=1 TYPE="Organization" SHORT="suborganization">
   <DEF-ARG POS=2 TYPE="Organization" SHORT="superorganization">
</DEF-RELATION>
```

Note that the SHORT attribute can help identify which organization fits in which argument.

While many relations are between two instances, some are between an instance and a data type value. For example, the publicationDate relation is between a publication and its date of publication, where date is a basic SHOE data type. This relation is written in SHOE as:

```
<DEF-RELATION NAME="publicationDate" SHORT="was written on">
   <DEF-ARG POS=1 TYPE="Publication">
   <DEF-ARG POS=2 TYPE=".DATE">
</DEF-RELATION>
```

An important consideration is to make the arguments of the relations as general as possible while still being correct. For example, we could have given subOrganization arguments of type Department and University. However, this would not capture the full meaning of the relation. Since both Department and University are subcategories of Organization, our original definition can be used to relate instances of these types, as well as many other different types.

This desire for broad applicability of relations is why we have not mentioned relations for the names of organizations or publications yet. Objects of both categories can have names, and in fact anything can have a name. Recall that the base ontology has a name relation, whose arguments are Entity and the STRING data type. Since all of our top-level computer science categories are subcategories of SHOEEntity, and SHOEEntity is a subcategory of Entity, we can use the name relation to provide a name for any instance of any category in our ontology. However, since name is such an important relation, it would be convenient to have a local version of it. SHOE's DEF-RENAME element can be used for this purpose. For example:

```
<DEF-RENAME FROM="base.name" TO="name">
```

This creates a local name for the name relation, so that within the context of the CS Department ontology, it can referred to as name, instead of base.name.

```
<DEF-INFERENCE DESCRIPTION="Transitivity of Suborganizations">
<INF-IF>
   <RELATION NAME="subOrganization">
        <ARG POS="FROM" VALUE="x" USAGE="VAR">
        <ARG POS="TO" VALUE="y" USAGE="VAR">
   </RELATION>
   <RELATION NAME="subOrganization">
        <ARG POS="FROM" VALUE="y" USAGE="VAR">
        <ARG POS="TO" VALUE="z" USAGE="VAR">
   </RELATION>
</INF-IF>
<INF-THEN>
   <RELATION NAME="subOrganization">
        <ARG POS="FROM" VALUE="x" USAGE="VAR">
        <ARG POS="TO" VALUE="z" USAGE="VAR">
   </RELATION>
</INF-THEN>
</DEF-INFERENCE>
```

Figure 6.1: The “transitivity of suborganizations” rule.

The final step in building the ontology is to provide a set of inference rules that help constrain the possible meanings of the terms and allow reasoners to infer implicit information from a set of assertions. We defined three inference rules for this ontology. These rules state that subOrganization is a transitive relation, that affiliatedOrganization is a symmetric relation, and that membership in and organization transfers through the subOrganization relations. A SHOE example of the transitivity of subOrganization is given in Figure 6.1

When the ontology was completed, it was embedded in an HTML page. This page also included a human-readable description of the ontology that serves as a handy reference for users to learn about the ontology. The ontology page was made publicly available via a web server so that it would be accessible to all web-based agents and could be reused by other people.

6.1.2 Annotating Web Pages

After an ontology has been created, it is possible to begin annotating web pages with SHOE content. The categories of the CS Department ontology have a direct correspondence with the main topics of many web pages, and thus can be used to describe those web pages. For example, it can used to describe department homepages, faculty homepages, student homepages, research project homepages, course homepages and publication lists.

We began by annotating the pages of the Parallel Understanding Systems (PLUS) research group. These pages included a group homepage, a members page, a publications page, a projects page, and a software page. We will use the group homepage (shown in Figure 6.2) to describe a simple example of SHOE markup. The first step is to identify the instances mentioned on the web page. Since this is a homepage, the most important instance is the subject of the page, which is the PLUS group. Other instances mentioned include the Department of Computer Science, the University of Maryland, the High Performance Systems Software Lab, and the Advanced Information Technology Lab.

A fundamental problem in distributed systems is knowing when data from different sources describes the same instance. The SHOE approach requires that each instance have a unique key. A URL can often serve as this key because it identifies exactly one web page, which is owned by a single person or organization. If an instance has a homepage, then the URL of this page is a good candidate for the key. Thus, the key for the PLUS group instance is *http://www.cs.umd.edu/projects/plus/*. This was used as the reference instance for the PLUS Group page, as shown in Figure 6.3. Every reference instance must also commit to at least one ontology by means of a USE-ONTOLOGY tag. For this application, the ontology is the CS Department

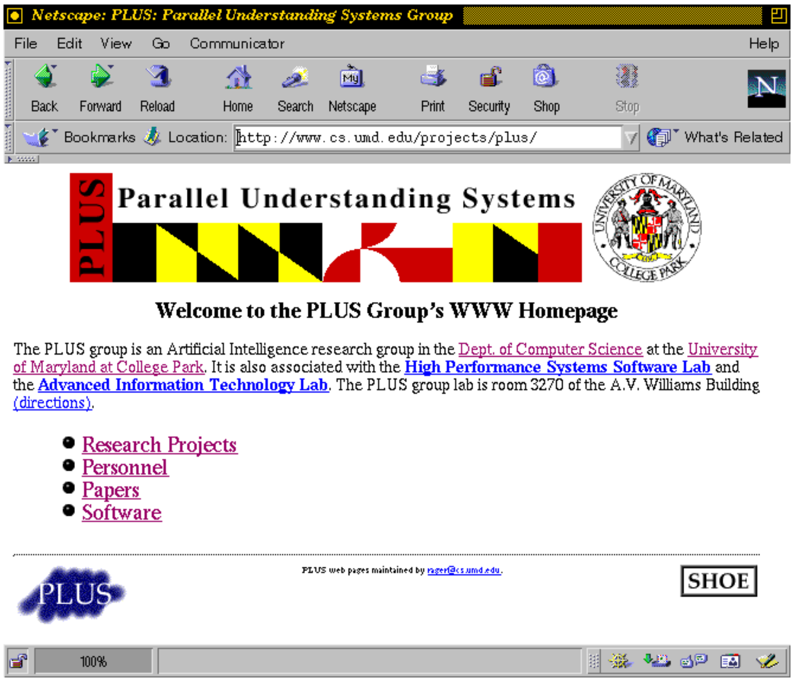

Figure 6.2: The PLUS group's homepage.

ontology.

The next step is to classify each of the instances according to the ontology. The PLUS group is a research group, and thus was categorized under cs.ResearchGroup as shown in Figure 6.3. We also categorized the Department of Computer Science as a cs.Department and the University of Maryland as a cs.University. Note the keys for these instances were chosen based on the URLs of their homepages, which are available from the hypertext links in the PLUS page (they are the values of HREF attributes in A tags).

Finally, we identified relations between the instances. Obviously there were relationships between the PLUS group and the other instances mentioned on the page. The page's text says that the PLUS group is associated with the High Performance Systems Software Lab and the Advanced Information Technology Lab. We decided

```
<INSTANCE KEY="http://www.cs.umd.edu/projects/plus/">
<USE-ONTOLOGY ID="cs-dept-ontology" VERSION="1.0" PREFIX="cs"
       URL="http://www.cs.umd.edu/projects/plus/SHOE/onts/cs.html">
<CATEGORY NAME="cs.ResearchGroup">
<CATEGORY NAME="cs.School" FOR="http://www.umd.edu/">
<CATEGORY NAME="cs.Department" FOR="http://www.cs.umd.edu/">
<RELATION NAME="cs.affiliatedOrganization">
   <ARG POS="TO" VALUE="http://www.cs.umd.edu/users/hendler/AITL/">
</RELATION>
<RELATION NAME="cs.affiliatedOrganization">
   <ARG POS="TO" VALUE="http://www.cs.umd.edu/projects/hpssl.html">
</RELATION>
<RELATION NAME="cs.subOrganization"
   <ARG POS="FROM" VALUE="http://www.cs.umd.edu/">
   <ARG POS="TO" VALUE="http://www.umd.edu/">
</RELATION>
<RELATION NAME="cs.subOrganization">
   <ARG POS="TO" VALUE="http://www.cs.umd.edu/">
</RELATION>
<RELATION NAME="cs.name">
   <ARG POS="TO" VALUE="Parallel Understanding Systems">
</RELATION>
</INSTANCE>
```

Figure 6.3: SHOE markup for the PLUS group.

that here "associated with" meant "affiliated organization of" and created cs.affiliatedOrganization relations between the PLUS group and each of these organizations. Note that once again, the URLs from the hypertext links were chosen as the keys for these instances. The page also mentions that the research group is "in the Dept. of Computer Science at the University of Maryland." This clearly indicates parent organizations of the group, and indicates that the University of Maryland is the parent organization of a particular computer science department. We could have created cs.subOrganization relations for all three relationships, but due to the transitivity of the relation (as specified in the ontology), the fact that the PLUS group is a suborganization of the University of Maryland is redundant with the other two. Note that in the figure, whenever the PLUS group was the subject of a relation assertion, the corresponding argument was omitted because in SHOE, the argument's value defaults to the reference instance's key.

When the SHOE tags were completed they were inserted into the BODY of the web page. However, since web browsers ignore any tags that they do not recognize, and SHOE content is entirely contained within such tags, there is no change in the presentation of the pages. Thus, the tags only affect software that is SHOE aware.

Using the methodology described above, we added SHOE markup to the remaining pages of the PLUS group and to the homepages of its members. Since the pages were already on the Web, the annotations immediately became publicly available. Most of these annotations were produced using the Knowledge Annotator, although text editors were used in some cases. However, it became clear that both approaches were somewhat time-consuming, and that it would take a long time to annotate a large corpus using these techniques.

6.1.3 Generating Markup on a Large Scale

It was clear that creating a sufficient amount of markup to test the Semantic Web idea would either require a large number of people to perform the task manually, or the use of specialized tools. In this section we will discuss the use of Running SHOE to create computer science markup, and the use of another program to extract publication information from a useful website that was unsuitable for Running SHOE.

Computer science department web sites often contain pages that have a semi-regular structure, such as faculty, project, course, and user lists. Often, the items in these list contain an <A> tag that provides the URL of the item's homepage, and this element's content is the name of the entity being linked to, providing us with a value for the *name* relation. Other properties of the instance are near the <A> tag and are delimited by punctuation, emphasis, or special spacing. These kinds of pages are ideal for Running SHOE, which was described in Section 5.2.3.

For example, consider the page of computer science faculty at the University of Maryland. As shown in Figure 6.4, the HTML used to describe each faculty member has a standard format. The person's information is preceded by a <DT> tag, and followed by a pair of newlines. The HREF attribute of the A element is the URL of their homepage, which serves as a good instance key for them, while the content of the element is their name. The beginning and ending of the list of faculty is indicated by the <DL> and </DL> tags. We can create three SHOE assertions for each item in the list. We know that the category of every instance is cs.Faculty, we can extract the cs.name for each instance, and we know that each instance is a cs.member of the Department of Computer Science at UMCP. The screen shot in Figure 5.4 shows the use of Running SHOE on this particular page.

Using Running SHOE, a single user created SHOE markup for the faculty,

```
<HR>

<DL>
<DT><A HREF="http://www.cs.umd.edu/~agrawala/">Ashok K. Agrawala</A>
<DD>Professor, CS and UMIACS. IEEE Fellow. Ph.D., Harvard
University, 1970.
<DD><I>Research Interests:</I> Design and evaluation of systems,
real time systems, networks.

<DT><A HREF="http://www.cs.umd.edu/~yiannis/">John (Yiannis)
Aloimonos</A>
<DD>Professor, CS, CfAR, and UMIACS. NSF Presidential Young
Investigator. Ph.D., University of Rochester, 1987.
<DD><I>Research Interests:</I> Artificial intelligence, vision,
robotics, learning, neuro-informatics.

<DT><A HREF="http://www.cs.umd.edu/~waa/">William A. Arbaugh</A>
<DD>Assistant Professor, CS and UMIACS. Ph.D., University of
Pennsylvania, 1999.
<DD><I>Research Interests:</I> Information System Security, Embedded
Systems, Operating Systems, and Networking.

...

<DT><A HREF="http://www.cs.umd.edu/~mvz/">Marvin V. Zelkowitz</A>
<DD>Professor, CS and UMIACS. Co-Dir., Fraunhofer Center --
Maryland. IEEE Fellow. Ph.D., Cornell University, 1971.
<DD><I>Research Interests:</I> Software engineering, environment
design, program complexity and measurement.

</DL>
```

Figure 6.4: Source HTML for the University of Maryland's CS faculty page.

courses, and projects of 15 major computer science departments in less than a day. Although, an experienced user can create markup for a single page in mere minutes, Running SHOE is only effective when there are long lists with regular structure. However, there are many important resources from which it cannot extract information, such as CiteSeer (*http://citeseer.nj.nec.com/cs*), an index of online computer science publications. 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 site.

To extract SHOE from CiteSeer, we built a tool called Publication SHOE Maker (PSM). 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 (BibTex is a common format for bibliographic 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.

In the examples described in this section, we created SHOE markup for pages that were owned by other parties. This raises a significant question: if an annotator does not have write privileges for the annotated web pages, then how does he associate SHOE assertions with them? The answer is create new web pages on his or her server that serve as summaries or indexes of the original web pages. Each of these pages contain a reference instance whose key is the URL of the page from which the markup was extracted. They may consist solely of SHOE tags, or may include additional summary information in HTML.

Since the results of Running SHOE and PSM are no different from those of text editors or the Knowledge Annotator, they can be processed by SHOE agents in the same manner. Since SHOE allows individuals to annotate their own pages, or make assertions about the content of other web pages, the efforts of information providers and professional indexers can be combined.

6.1.4 Processing the Markup

The main goal for the CS Department application was to use SHOE to improve web search of CS Department information. To achieve this, we chose to implement a repository-based access system that relied on the Exposé web-crawler. In order to compare the features of different repositories, we configured the web-crawler to store its results in both XSB and Parka KBs. Exposé was able to gather 38,159 assertions from the various web pages that we annotated. Once the knowledge was loaded into the KBs, the SHOE Search tool could be used to query them, either as a stand-alone application or as a Java applet.

The possible benefits of a system such as this one are numerous. A prospective student could use it to inquire about universities that offered a particular class or performed research in certain areas. Or a researcher could design an agent to search for articles on a particular subject, whose authors are members of a particular set of institutions, and were published during some desired time interval. Additionally, SHOE can combine the information contained in multiple sources to answer a single query. For example, to answer the query "Find all papers about ontologies written by authors who are faculty members at public universities in the state of Maryland" one would need information from university home pages, faculty listing pages, and publication pages for individual faculty members. Such a query would be impossible for current search engines because they rank each page based upon how many of the query terms it contains.

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. The source of this problem was that the sites occasionally use different URLs to refer to the same person, and thus the SHOE assertions used different instance keys for the same entity. 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. This problem is an important one for future work.

6.1.5 Revising the CS Department Ontology

The CS Department ontology described in Section 6.1.1 is adequate for the purpose of representing and querying the most common concepts that are relevant to its domain. However, it was not designed to promote reusability and interoperability. For example, the categories Person and Organization are more general than the domain of computer science departments, and could be useful in many other ontologies. Yet, for an ontology to reuse these terms, it would have to include the entire CS Department ontology. Clearly, the CS Department ontology should have been more modularized. In this section we will discuss the development of a more modular set of ontologies, and describe how the CS Department ontology can be revised in such a way that any web pages that depend on it do not need to be updated. This last point is critical on the Semantic Web, because the web pages that commit to an ontology may be distributed across many servers and owned by many different parties, making a coordinated update impossible.

We begin by creating a general ontology, called general-ont, that contains elements that are common to most web domains. The top-level categories of this ontology are Agent, PhysicalObject, Event, Location, Address, Activity, and WebResource. These categories are refined with subcategories, including the Person, Organization, and Work classes from the CS department ontology. The ontology has some of the relations originally defined in the CS department ontology, including member and head. It also corrects one source of confusion in the CS ontology by using the relation subOrganizationOf instead of subOrganization; this new name more clearly signifies that the first argument is intended to be the child organization. Finally, the ontology includes all of the CS Department ontology's inference rules, and adds additional ones, to state conditions such as "A person who works for an organization is a member of that organization" and "A person who is the head of an organization is a member of that organization."

Another area of the CS Department ontology that could be reused by many other ontologies is the Publication taxonomy and relations. These can be contained in a document ontology called document-ont. The top-level category of this ontology is Document, and its subcategories include unpublished documents as well as published ones.

Continuing with our modularization of the CS Department ontology, it is clear that none of the original ontology is specific to CS Departments. In fact, most of the remaining ontology is applicable to universities in general or other types of academic departments. Therefore, we must also create a university ontology, called university-ont. This ontology extends both general-ont and document-ont, and includes cs-dept-ontology categories such as Faculty, Student, University and Department.

Finally, we can revise the CS Department ontology so that it is integrated with our new, modularized ontologies. We will call the new ontology version 1.1 of the cs-dept-ontology. This ontology extends both university-ont and document-ont and, for reasons we will discuss momentarily, is backward-compatible with cs-dept-ontology, version 1.0. All of the categories and relations are defined in other ontologies, but to maintain backward-compatibility, we need local names for these components. Therefore, the ontology consists of a series of DEF-RENAME definitions. In some cases, the ontology needs to contain components just to ensure backward-compatibility. For example, recall that in general-ont, we decided to change the name of subOrganization to subOrganizationOf. For cs-dept-ontology, version 1.1 to be compatible with version 1.0, it must rename subOrganizationOf to subOrganization.

Now that we have a backward-compatible revision of the CS Department ontology, the web pages that committed to version 1.0 should be integrated with two compatible ontology perspectives, one based on version 1.0, and one based on version 1.1. What this means is that the old web pages can be automatically integrated with web pages based on the new ontology from the perspective based on version 1.1. Thus, there is no need to upgrade all of the web pages at once. Instead, their owners can upgrade at their own pace (by committing to the new version of the ontology and possibly using some of the new terms) or even choose not to upgrade.

The need to modularize or restructure ontologies after they have come into use will be important on the Semantic Web. Although not all revisions will be as drastic as the one described in this section, SHOE's backward-compatibility feature and compatible ontology perspectives ensure that revisions can occur smoothly.

6.1.6 Summary

We have described a case study in which SHOE was applied to the domain of computer science departments. We demonstrated how a simply SHOE ontology can be constructed and how web pages can be annotated. We showed that there were many possible means of acquiring SHOE information, including manual human annotation and semi-automated wrapper generation, and that it was possible to generate a significant number of SHOE assertions in days. We also examined how the basic architecture discussed in Section 5.1.5 could be used in practice and compared the use of two different knowledge base systems as repositories for SHOE knowledge. Finally, we discussed how ontologies could be modularized, even after they have been put to practical use, and how backwards-compatibility can eliminate the need to upgrade existing web pages when an ontology must be revised.

6.2 Food Safety

The second case study in this chapter describes the application of SHOE to the domain of food safety. Whereas the first case study was carefully constrained, this one focused on a real-world problem and was performed in concert with the Joint

Institute for Food Safety and Applied Nutrition (JIFSAN). JIFSAN is a partnership between the Food and Drug Administration (FDA) and the University of Maryland, and is working to expand the knowledge and resources available to support risk analysis in the food safety area. One of their goals is to develop a website that will serve as a clearinghouse of information about food safety risks. This website is intended to serve a diverse group of users, including researchers, policy makers, risk assessors, and the general public, and thus must be able to respond to queries where terminology, complexity and specificity may vary greatly. This is not possible with keyword-based indices, but can be achieved using SHOE. The work described in this section was first presented in an earlier paper [51].

In order to scope the project, JIFSAN decided to focus the SHOE effort on a specific issue of food safety. The chosen issue was Transmissible Spongiform Encephalopathies (TSEs), which are brain diseases that cause sponge-like abnormalities in brain cells. “Mad Cow Disease,” which is technically known as Bovine Spongiform Encephalopathy (BSE), is the most notorious TSE, mainly because of its apparent link to Creutzfeldt-Jakob disease (CJD) in humans. Recent Mad Cow Disease epidemics and concerns about the risks BSE poses to humans continue to spawn international interest on the topic.

6.2.1 The TSE Ontology

Unlike the CS Department ontology, the TSE ontology was created by a team of knowledge engineers and domain experts from the FDA and the Maryland Department of Veterinary Medicine. The ontology focused on the three main concerns for TSE Risks: source material, processing, and end-product use. The top-level categories in the source material area were Material, Animal, DiseaseAgent, and two important subcategories of Material were Tissue, and BodyFluid. The processing area had a top-level category called Process, which was subdivided into BasicProcess and AppliedProcess. The BasicProcess category was further subdivided based on the nature of process, while AppliedProcess was subdivided based on the purpose of the process. The end-product use portion of the ontology had EndProduct (which was a subcategory of Material), ExposureRoute, and Risk categories. Additional basic categories such as Person, Organization, and Event rounded out the rest of the ontology, which has 73 categories in all.

After creating the categories, it was necessary to create the relations. Relations for Material included weight and volume, which were ternary because they had to relate a specific material to a quantity and express the measurement unit of the quantity. Relations for DiseaseAgent included its transmissibility, transmission method, and symptoms. For Process, the important relations included hasInput and hasOutput, which identify its inputs and outputs, and duration, which identified the length of the process. A total of 88 relations were created, twelve of which had three or more arguments.

A significant problem in the design of this ontology was scoping the effort. Many of the categories and relations that were developed were not of use to the subsequent annotation effort. Grüninger [43] suggests that competency questions can be used to scope an ontology and later test it for completeness. A competency question is essentially a question about the domain that the ontology should be able to answer. For a Semantic Web markup language, we suggest that an ontology also be scoped by the kinds of information that is currently (or will soon be) available on web pages. Otherwise, portions of the ontology may be irrelevant to the annotation process.

6.2.2 Food Safety Annotation

Following the creation of the initial ontology, the JIFSAN team annotated web pages. This process can be divided into two distinct markup activities. First, since the Web had little information on animal material processing, we created a set of pages describing many important source materials, processes and products, and added annotations to those pages. Second, we annotated pages that provided general descriptions of the disease, described recommendations or regulations, or presented experimental results.

The web pages created by the team were very easy to annotate, since their primary purpose was to provide useful SHOE markup. Pages were created to describe sources such as cattle and pigs; processes such as slaughter, butchering, and rendering; and end products such as animal feed, human food, and personal products. The SHOE annotations for each process included its category, inputs, and outputs. Annotations for the other pages were typically just the category.

The second set of web pages were much more difficult to annotate. Unlike the computer science department web pages of the other case study, these pages were not homepages and had very few hypertext links. As such, it was difficult to find instances that had obvious keys. Instead, the user had to identify a significant noun and then create a key for it. Furthermore, the pages were typically long, prose texts, instead of short, structured forms. Thus extracting relationships typically involves parsing of sentences, and since natural language is much richer than knowledge representation languages, the process of translating sentences into suitable structures was difficult. Prose documents can potentially be translated into numerous SHOE assertions, and the extent of markup required for such pages was unclear. To compound matters, the important concepts of these pages often had little overlap with the original ontology.

A better strategy would have been to decide how such pages would be annotated and design the ontology to facilitate this kind of markup. Such a strategy would of course depend on the intended use for the markup. In the case of these pages, improving search was the primary purpose, and thus markup about the document's source and data, and a subject matter classification would probably have been sufficient.

6.2.3 Processing the Annotations

The basic architecture for the food safety domain is the one described in Section 5.1.5. Although the system has not been officially released yet, it is intended that various sources outside of JIFSAN will be able to annotate their web pages with the TSE ontology and thus be accessible via SHOE query tools. The basic process would work as follows:

1. Knowledge providers who wish to make material available to the TSE Risk Website use the Knowledge Annotator or other tools to add SHOE markup their pages. The instances within these pages are described using elements from the TSE Ontology.

2. The knowledge providers then place the pages on the Web and notify JIFSAN.

3. JIFSAN reviews the site and if it meets their standards, adds it to the list of sites that Exposé, the SHOE web-crawler, is allowed to visit.

4. Exposé crawls along the selected sites, searching for more SHOE annotated pages with relevant TSE information. It will also look for updates to pages.

5. SHOE knowledge discovered by Exposé is loaded into a Parka knowledge base. Currently, XSB is not needed because the initial version of the TSE ontology

did not define inference rules. However, alternate KBs may be added later.

6. Java applets on the TSE Risk Website access the knowledge base to respond to users' queries or update displays. These applets include the TSE Path Analyzer (described in Section 6.2.4) and SHOE Search.

It is important to note that new websites with TSE information will be forced to register with JIFSAN. This makes Exposé's search more productive and allows JIFSAN to maintain a level of quality over the data they present from their website. However, this does not restrict the ability of approved sites to get current information indexed. Once a site is registered, it is considered trusted and Exposé will revisit it periodically.

6.2.4 The TSE Path Analyzer

One of JIFSAN's requirements was the need to analyze how source materials end up in products that are eventually consumed by humans or animals. This information is extremely valuable when trying to determine the risk of contamination given the chance that a source material is contaminated. It is expected that information on each step in the process will be provided on different web sites (since different steps are usually performed by different companies), thus using a language like SHOE is essential to integrating this information.

To accommodate this need, we built the TSE Path Analyzer, which is an example of a domain specific query tool. The TSE Path Analyzer allows the user to pick any combination of source, process and product from lists that are derived from the taxonomies of the ontology. The system then displays a graph of all possible pathways that match the query. For example, Figure 6.5 displays the results of a query

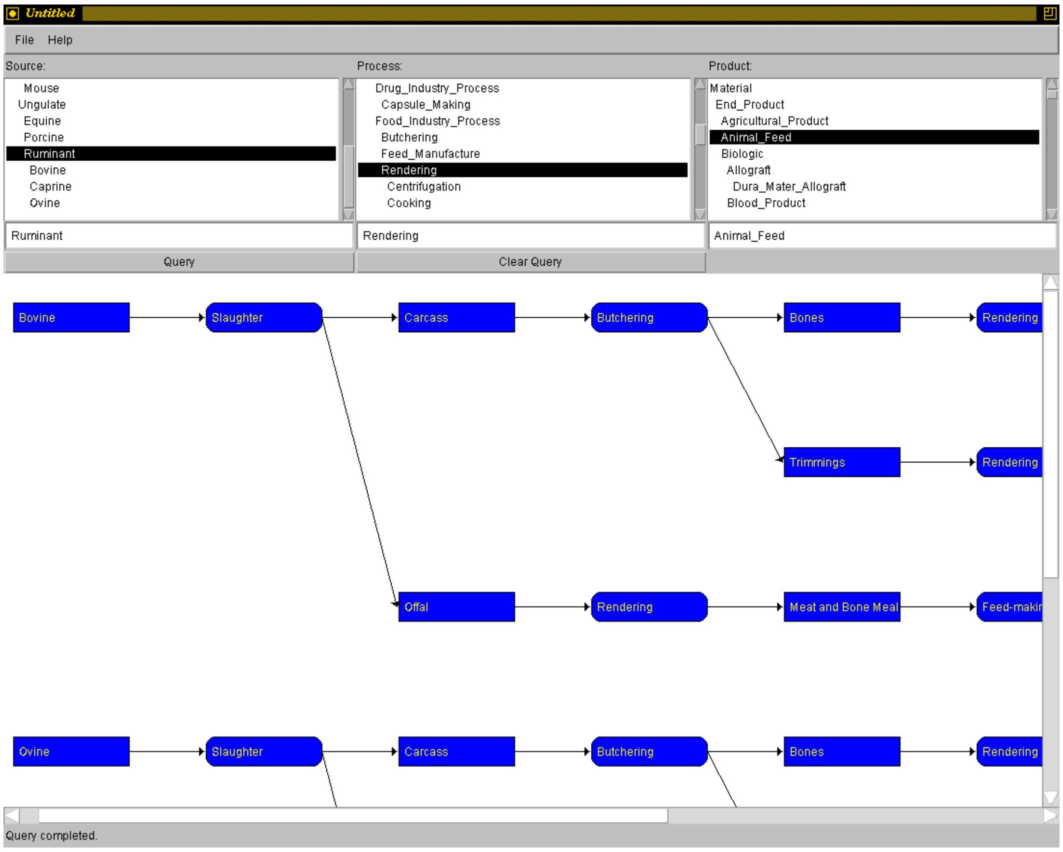

Figure 6.5: The TSE Path Analyzer.

to find all pathways that begin with ruminant source material, include a separation process, and result in a ruminant feed product. In the display, square boxes indicate source materials and rounded boxes indicate processes. The user can click on any box to open a web page with more details on that subject. This tool essentially provides users with a dynamic map of a set of web pages based upon a semantic relation.

The Path Analyzer is a repository-based query tool. It issues its queries to a Parka repository that is updated by the Exposé web-crawler. In response to a user's query, it grows the graph from the selected source material by retrieving all processes that have that material as an input, and then retrieving all products that are outputs of those processes. The process is repeated on the products until eventually the entire set of paths from the source material are computed. Then the graph is pruned to remove any paths that do not involve the selected process and end product. If no source is selected, then the procedure starts with the selected process and works it way through the process's inputs and outputs. If only a product is selected, then the procedure works its way back through the processes that created the product.

The biggest advantage of the TSE Path Analyzer is that it provides a simple user interface to solve a complicated problem. Users only have to select items from up to three list and press a button. A tool like the TSE Path Analyzer could not have been created without a markup language to provide its source information.

6.2.5 Summary

The food safety case study demonstrated some of the problems with applying SHOE to real-world problems. First, ontology design can be much more difficult, and significant effort needs to be devoted to properly scoping the ontology. Second, this application demonstrated that certain kinds of pages are more ideal for SHOE than others. The most effective use of SHOE is in describing web pages that have many hypertext links and some sort of structure (such as field lists or tables). Although, SHOE can be applied to prose-like pages, they require more work on behalf of the annotator. We also demonstrated how domain-specific tools can use SHOE to provide simple interfaces that help users solve complex problems.

Chapter 7

Comparison of Semantic Web Languages

Although SHOE was one of the first Semantic Web languages, there are other languages with different syntaxes, semantics, and approaches to the problems of knowledge representation on the Web. We will compare the most significant of these languages (Ontobroker, RDF, and OIL) to SHOE. We will then describe DAML+OIL an emerging standard that is the result of an international effort to combine the best features of SHOE and these languages.

7.1 Ontobroker

The Ontobroker system [31, 23] is similar to SHOE in many respects. It includes an ontology definition language, a web page annotation language, a web crawler, an inference engine, and a set of query interfaces. Ontobroker is based on frame-logic [57], and includes primitives for subclassing, instantiation, attribute declaration, attribute value specification, predicates, and rules. Frame-logic rules can be built using all of the connectives of predicate logic (implication, conjunction, disjunction, and negation), and variables may be universally or existentially quantified. As such, the language is more expressive than SHOE, but suffers from the scalability problems

```
Object[].
  Person :: Object.
    Faculty :: Person.
      Chair :: Faculty.
    Student :: Person.
  Organization :: Object
    Department :: Organization.

  Person[
    name =>> STRING;
    headOf =>> Organization].

  Faculty[
    teaches =>> Class;
    advises =>> Student].

  FORALL X, Y
    X:Chair <- X:Person[headOf ->> Y] AND Y:Department
```

Figure 7.1: An Ontobroker ontology.

of predicate logic. One advantage of a frame-logic language is that ontology objects (such as classes and attributes) are first class citizens and can be used in expressions.

Ontobroker ontologies are written in ordinary frame-logic instead of an SGML or XML syntax. This is because these ontologies are not meant to be shared on the Internet, but are instead tailored to homogeneous intranet applications. As such there is also no need for ontology inclusion or versioning. A short Ontobroker university ontology is shown in Figure 7.1. In frame-logic, $C_1 :: C_2$ means that class C_1 is a subclass of C_2, and $C[A =>> T]$ means that class C has an attribute A of type T. Ontobroker axioms can have existential or universal variables (with the EXISTS and FORALL quantifiers, and can use the logic connectives $<-$, $->$, $<->$, AND, OR, and NOT.

An Ontogroup is a set of users that agree to a particular domain-specific ontology.

This group serves as an index of content providers, which is used to focus the work of the web crawler. However, this means that new users must register with the group to begin providing information, and the annotations will be of little use to those who are not members of the community, since they do not have access to the ontology information.

The language for annotating web pages also makes use of frame-logic notation and is very compact. It can be embedded in ordinary HTML pages by way of a small extension to the common <A> tag. The class of an instance is specified by an expression of the form O:C, as demonstrated here:

```
<a onto=" 'http://www.state.edu/users/jsmith/':Chair">
```

An attribute value may be specified by an expression of the form $O[A - >> V]$, as in:

```
<a onto=" 'http://www.state.edu/users/jsmith/' [name='Jane Smith']">
```

Furthermore, a set of special key words allows the annotations to avoid redundancy with other information on the page. For example, the page keyword indicates that the value should be supplied by the URL of the page. Additionally, the href keyword specifies that the value is the same as the value of the href attribute in the same tag, and the body keyword specifies that the content of the tag supplies the value. For example, on the page *http://www.state.edu/users/jsmith/*, an alternate way of supplying the name annotation shown above is:

```
<a onto="page[name=body]">Jane Smith</a>
```

One of the best features of the Ontobroker approach is its compact and simple language for expressing both ontologies and data. An important aspect of this is the use of the special keywords page, href, and body to reduce redundancy between the markup and the document's content. These feature can help ensure that when a document changes, its markup stays synchronized with the content. However, Ontobroker is not meant to be used on the entire Web. Its ontologies are not publicly available, instead they are designed for local applications that process a controlled set of web pages. Although this allows Ontobroker to ignore many of the problems discussed in this thesis, it makes it unsuitable for use as a Semantic Web platform.

7.2 RDF

The Resource Description Framework (RDF) [62, 61] is a W3C recommendation that attempts to address XML's semantic limitations. It presents a simple model that can be used to represent any kind of data. This data model consists of nodes connected by labeled arcs, where the nodes represent web resources and the arcs represent properties of these resources. It should be noted that this model is essentially a semantic network, although unlike many semantic networks, it does not provide inheritance. The nodes/arcs model also means that RDF is inherently binary, however, this does not restrict the expressivity of the language because any n-ary relation can represented as a sequence of binary relations.

RDF can be exchanged using an XML serialization syntax, which is shown by example in Figure 7.2. The basic syntax consists of a Description element which contains a set of property elements. The about attribute identifies which resource is described. The property rdf:type is used to express that a resource is a member of a given class, and is equivalent to the *instance-of* link used in many semantic nets and frame systems. There are a number of abbreviated variations of the RDF syntax, which is an advantage for content providers but requires more complex RDF parsers. Using two of the abbreviation techniques, Figure 7.2 can be rewritten as shown in Figure 7.3. It is important to note that all of these syntaxes have a well-defined

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:g="http://schema.org/general#">
  <Description about="http://www.state.edu/users/jsmith">
     <type resource="http://schema.org/university#Chair" />
     <g:name>Jane Smith</g:name>
  </Description>
</RDF>
```

Figure 7.2: An RDF Instance.

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:g="http://schema.org/general#"
     xmlns:u="http://schema.org/university#">
  <u:Chair about="http://www.state.edu/users/jsmith"
           g:name="Jane Smith" />
</RDF>
```

Figure 7.3: An Abbreviated RDF Instance.

mapping into the RDF data model, and thus avoid some of the problems with representational choices in basic XML. Nevertheless, it is still easy to create different representations for a concept.

To prevent accidental name clashes between different vocabularies, RDF assigns a separate XML namespace [14] to each vocabulary (these vocabularies, called schemas, can be formally defined using RDF Schema as discussed below). This approach has two disadvantages. First, since namespaces can be used with any element and RDF schemas need not be formally specified, it is possible to write RDF statements such that it is ambiguous as to whether certain tags are RDF or intermeshed tags from another namespace. Second, namespaces are not transitive, which means that each RDF section must explicitly specify the namespace for every schema that is referenced in that section, even for schemas that were extended by a schema whose namespace has already been specified.

In addition to the basic model, RDF has some syntactic sugar for different collections of objects called containers. There are three types of containers: bag, sequence, and alternative. A bag is unordered, a sequence is ordered, and an alternative is a set of choices. Using a special aboutEach attribute, a document can make statements that apply to every element in a collection.

Perhaps RDF's most controversial and least understood feature is reification. The intent of RDF reification was to allow statements to be made about statements. This can be used to provide metadata about the statement, such as creator, effective date, etc., or to provide additional information such as a confidence factor. Reification involves describing a statement with four statements. These statements use the property type to classify a resource as a Statement and the properties subject, predicate, and object, to model the three parts of the statement. A reified statement does not assert the statement it describes. This allows providers to talk about statements without claiming they are true, although the mechanism is rather verbose. A statement can be both described and asserted by associating a bagId with an RDF description. In addition to asserting the statements, it creates a collection of corresponding reified statements, and descriptions can use the value of the bagId to describe these statements. Some of the problems with RDF reification include confusion about whether it can be used for modalities, and also the fact that there is no way to distinguish between two identical statements in different documents. The latter problem means that any statement about a reified statement in one document must also apply to all identical statements in other documents.

To allow for the creation of controlled, sharable, extensible vocabularies the RDF working group has developed the RDF Schema Specification [16]. RDF schema allows users to create schemas of standard classes and properties using RDF. For this purpose, the specification defines a number of classes and properties that have specific semantics. The rdfs:Class and rdfs:Property classes allow a resource to be typed as a class or property respectively, and properties can be used to describe these classes and properties. The property rdfs:subClassOf essentially states that one class is a subset of another, and is equivalent to the *is-a* link used in semantic networks and frame systems. With the rdfs:subClassOf property, schema designers can build taxonomies of classes for organizing their resources. RDF Schema also provides properties for describing properties; the property rdfs:subPropertyOf allow properties to be specialized in a way similar to classes, while the properties rdfs:domain and rdfs:range allow constraints to be placed on the domain and range of a property. An excerpt from the RDF Schema version of the university ontology from Figure 4.1 is given in Figure 7.4.

Note that since classes and properties are resources, they are identified by URIs. Each URI is the concatenation of the URL of the resource's source document, a hash ('#') and the resource's ID. For this reason, RDF does not have to handle problems of polysemy. Since URIs are often long and unwieldy, namespace prefixes can be used to create shorter identifiers. However, namespace prefixes cannot be currently used in attribute values, and thus some URIs must be written in full form.

In RDF, schemas are extended by simply referring to objects from that schema as resources in a new schema. Since schemas are assigned unique URIs, the use of XML namespaces guarantees that exactly one object is being referenced. Unfortunately, RDF does not have a feature that allows local aliases to be provided for properties and

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#">
   <rdfs:Class rdf:ID="Faculty">
      <rdfs:subClassOf rdf:resource=
            "http://schema.org/general#Person" />
   </rdfs:Class>

   <rdfs:Class rdf:ID="Student">
      <rdfs:subClassOf rdf:resource=
            "http://schema.org/general#Person" />
   </rdfs:Class>

   <rdfs:Class rdf:ID="Chair">
      <rdfs:subClassOf rdf:resource="#Faculty" />
   </rdfs:Class>

   <rdfs:Property rdf:ID="advises">
      <rdfs:domain rdf:resource="#Faculty" />
      <rdfs:domain rdf:resource="#Student" />
   </rdfs:Property>
</rdf:RDF>
```

Figure 7.4: An example university RDF schema.

classes. Although an alias can be approximated using the rdfs:subClassOf or rdfs:subPropertyOf properties to state that the new name is a specialization of the old one, there is no way to state an equivalence. This can be problematic if two separate schemas “rename” a class, because when schemas simply subclass the original class, the information that all three classes are equivalent is lost.

RDF schema is written entirely in RDF statements. Although at first this may seem like elegant bootstrapping, closer inspection reveals that it is only a reuse of syntax. RDF is not expressive enough to define the special properties rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain and rdfs:range, and thus correct usage of these properties must be built into any tool that processes RDF with RDF schemas.

A significant weakness of RDF is that it does not specify a schema inclusion feature. Although namespaces allow a document to reference terms defined in other documents, it is unclear as to whether the definitions of these terms should be included. In fact, it is unclear what constitutes the definition of a term. The problem is that the definition of a class (or property) is a collection of RDF statements about a particular resource using properties from the RDFS namespace. Typically, these statements appear on a single web page, grouped using an rdf:Description element. However, since a resource is identified by a URI, there is no reason why some of these statements could not appear in another document. Thus anyone could add to the definition of an object introduced in another schema. Although there are many situations where this is beneficial, accidental or malicious definitions may alter the semantics in an undesirable way. For example, someone could make the class WebDeveloper a subclass of OverpaidPerson, and anyone who stated that they were a WebDeveloper, would now also be implicitly stating they were an OverpaidPerson.

A possible solution to this is to treat XML namespaces as the only source of ontology inclusions, and then use extended ontology perspectives as described by Definition 3.22. However, this approach seems somewhat ad hoc, and may require that namespaces be added to include documents that have no new names, but instead provide only additional definitions for names from other namespaces.

RDF does not possess any mechanisms for defining general axioms, which are used in logic to constrain the possible meaning of a term and thus provide stronger semantics. Axioms can be used to infer additional information that was not explicitly stated and, perhaps more importantly for distributed systems such as the Web, axioms can be used to map between different representations of the same concepts. Useful axioms might specify that the subOrganization property is transitive or that the parentOf and childOf properties are inverses of each other. Many RDF proponents believe that axioms can be added to RDF by layering a logic language on top of RDF schema. However, it appears that doing so would result in an awkward syntax. The problem is that if the terms of complex logical formula are modeled resources, then RDF requires that each term be treated as an assertion, which would be incorrect. The only other option is to reify each term, which will not assert them, but since reification requires four statements to model each term, this is an extremely verbose and unwieldy method to represent logic sentences.

Another potential problem for RDF is the Web's tendency towards rapid change. Although RDF provides a method for revising schemas, this method is insufficient. Essentially, each new version of a schema is given its own URI and thus can be thought of as a distinct schema in and of itself. However, the revision is really just a schema that extends the original version; its only link to the original schema is by use of the rdfs:subClassOf and rdfs:subPropertyOf properties to point to the original definitions of each class and property. As such, a true equivalence is not established between the items. Additionally, if schemas and resources that refer to the schema that was updated wish to reflect the changes, they must change every individual reference to a schema object to use the new URI. Finally, since schemas do not have an official version associated with them, there is no way to track the revisions of a schema unless the schema maintainer uses a consistent naming scheme for the URIs.

Even with RDF Schema, RDF has very weak semantics. Still, there are many who believe that it provides a good foundation for interchanging data and that true semantic web languages can be layered on top of it. By layering, we mean creating a language that uses the RDF syntax, but also adds new classes and properties that have specific semantics. In the next two sections, we will discuss two languages that layer on top of RDF and RDF Schema.

7.3 OIL

OIL [32, 22, 33], which stands for Ontology Interchange Language or Ontology Inference Layer, is a language for describing ontologies on the Web. OIL's semantics are based on description logics, but its syntax is layered on RDF. One of the design goals for OIL was to maximize integration with RDF applications. Thus, most RDF Schemas are valid OIL ontologies, and most OIL ontologies can be partially understood by RDF processors. Unlike RDF, OIL has a well-defined semantics.

There are multiple layers of OIL, where each subsequent layer adds functionality to the previous one. Core OIL is basically RDFS without reification, which was omitted because as discussed in Section 7.2 it can be problematic. Standard OIL adds a number of description logic primitives to the Core OIL layer, and is the focus of most OIL work to date. Instance OIL adds the capability to model instances, essentially using RDF to describe the instances. Finally, Heavy OIL is an undefined layer that will include future extensions to the language. This layered approach allows applications to use pre-defined subsets of the language to manage complexity.

OIL starts with the basic primitives of RDF, classes and properties. There are two basic types of classes: primitive classes and defined classes. Primitive classes are essentially ordinary RDFS classes, while defined classes provide necessary and sufficient conditions for membership. Defined classes require the use of class expressions, which are boolean combinations of classes and slot constraints. The standard boolean operations are provided by oil:AND, oil:OR, and oil:NOT. Slot constraints restrict classes to only those instances which are the domain of a property where the range satisfies some constraint. Slot constraints include oil:HasValue, oil:ValueType, oil:MaxCardinality and oil:MinCardinality. The oil:HasValue constraint states that there must exist at least one value for the slot that is a member of a specified class expression. The oil:ValueType constraint states that all values for the slot must be members of a specified class expression. The cardinality constraints state that there must exist at most (or at least) n instances that have the value for the particular slot. In all of these cases, the oil:hasProperty property is used to indicate the property to which the constraint applies, and oil:hasClass is used to indicate the class expression (if any) of the constraint. An example OIL class definition for Husband is given in Figure 7.5. In this ontology, a husband is a male who is married to a female.

OIL slots are RDF properties, and thus slot definitions can use RDFS constructs such as rdfs:subclassOf, rdfs:domain, and rdfs:range. OIL also adds properties and classes that can be used to give slots more precise definitions. The oil:inverseRelationOf property states that two properties are inverse relations. If a property is an instance of the oil:TransitiveProperty class, then the property is

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
      xmlns:oil="http://www.ontoknowledge.org/oil/rdfs-schema" >

<oil:DefinedClass rdf:ID="Husband">
   <rdfs:subClassOf rdf:resource="#Male" />
   <oil:hasSlotConstraint>
      <oil:HasValue>
         <oil:hasProperty rdf:resource="#isMarriedTo" />
         <oil:hasClass rdf:resource="#Female" />
      </oil:HasValue>
   </oil:hasSlotConstraint>
</rdfs:Class>
</rdf:RDF>
```

Figure 7.5: An OIL ontology that defines the class Husband.

transitive. Finally, if a property is a subclass of oil:SymmetricProperty, then it is a symmetric relation.

In addition to defining classes and slots, OIL ontologies can describe themselves with metadata, import modules and provide a rule base. There are a standard set of meta-properties, based on the Dublin Core, that include ontology name, ontology author, and others. The import mechanism is simply to use XML namespaces, and suffers from the same drawbacks as RDF in this area. The rule base is intended to provide additional axioms or global constraints for the ontology, but its structure is currently undefined.

The advantages of OIL are tied to its description logic basis. If two ontologies used the same set of base terms in their definitions, then it is possible to automatically compute a subsumption hierarchy for the combination of the ontologies. Additionally, the rich modeling constructs allow consistency to be checked, which eases the construction of high-quality ontologies. However, it is possible for logical inconsistencies to arise due to instances, which will be distributed across the Semantic Web and thus harder to control. There are no guidelines as to how reasoners should approach this kind of inconsistency. For example, if a Person class defined the oil:maxCardinality of a marriedTo slot to be one, what should happen if different documents contain assertions about different people being married to Madonna? OIL's other weaknesses are inherited from RDF. It has no explicit import mechanism and inadequate support for ontology evolution. OIL also cannot specify many of the common kinds of articulation mappings needed to integrate ontologies. For example, OIL cannot express synonymy of classes or properties, and cannot express mappings between different structures that represent the same concept. For these reasons, it seems that OIL is better suited as an RDF representation of description logic than as a foundation for the Semantic Web.

7.4 DAML

The DARPA Agent Markup Language (DAML) [54, 55] is perhaps the highest profile Semantic Web language. This high profile is in part because DAML is a major DARPA project with multiple academic and industry teams involved, and in part because it involves many member of the W3C, including Tim Berners-Lee himself. DAML attempts to combine the best features of other Semantic Web languages, including RDF, SHOE, and OIL. The earliest versions of DAML were officially called DAML-ONT, but a later effort to more closely involve the developers of OIL has resulted in DAML+OIL. In the rest of this section, unless explicitly stated otherwise, the term DAML refers to DAML+OIL.

Like OIL, DAML builds upon RDF and has a description logic basis. DAML allows class expressions to be a single class, a list of instances that comprise a class, a property restriction, or a boolean combination of class expressions. The daml:intersectionOf, daml:unionOf, and daml:complementOf properties provide conjunction, disjunction, and negation of class expressions, and thus serve the same purpose as the oil:AND, oil:OR, and oil:NOT classes.

A DAML property restriction is indicated by the daml:Restriction class, which contains a daml:onProperty property that specifies the slot being restricted, as well as additional information about the restriction. The daml:toClass property is used to say that all values for the slot must be members of the specified class expression, and has the same use as the oil:ValueType class. The daml:hasClass property is like the oil:HasClass class, and states that at at least one value for the slot must be a member of the specified class expression. DAML also has a daml:hasValue property, which does not have an equivalent in OIL. This property is used to state that one value of the slot must equal the specified value. Like OIL, DAML includes cardinality restrictions, specifically daml:minCardinality and daml:maxCardinality. Because of the different way that DAML structures restrictions, it also needs daml:hasClassQ, daml:maxCardinalityQ, and daml:minCardinalityQ properties so that cardinality restrictions can be qualified by a specific class expression. Figure 7.6 presents the DAML version of the Husband ontology from Figure 7.5.

DAML also provides primitives for defining properties. In addition to the basic ones available in RDF, it adds a daml:inverseOf property and daml:TransitiveProperty class, which are identical to elements of OIL. DAML also has the daml:UniqueProperty and daml:UnambiguousProperty classes which state that a property can only have one value per instance and that a value can only belong to one instance, respectively.

Like SHOE, DAML has an explicit feature for including ontologies, provides a

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
      xmlns:daml="http://www.daml.org/2001/03/daml+oil#" >

<daml:Ontology rdf:about="">
   <daml:versionInfo>1.0</daml:versionInfo>
   <daml:imports rdf:resource="http://schema.org/base#" />
</daml:Ontology>

<daml:Class rdf:ID="Husband">
   <rdfs:subClassOf rdf:resource="#Male" />
   <rdfs:subClassOf>
      <daml:Restriction>
         <daml:onProperty rdf:resource="#isMarriedTo" />
         <daml:hasClass rdf:resource="#Female" />
      </daml:Restriction>
   </rdfs:subClassOf>
</daml:Class>

</rdf:RDF>
```

Figure 7.6: A DAML ontology that defines the class Husband.

means for handling synonymous terms, and provides some primitive version information. DAML's daml:imports is similar to SHOE's USE-ONTOLOGY element. It is transitive and specifies that the definitions from the imported document also apply to the current ontology. This allows extended ontology perspectives (see Definition 3.22) to be used with DAML. However, unlike SHOE, DAML uses XML namespaces to provide names, which requires some redundancy between the namespace declarations and daml:imports statements.

DAML has a daml:equivalentTo property that is used to state that two resources are identical. Since classes and properties are resources, this can be applied to them as well as other classes. Thus, daml:equivalentTo provides the functionality of SHOE's DEF-RENAME element, plus the capability to state that two instances are identical. The latter feature is extremely useful on the Web, where resources are identified by URLs. Since syntactically different URL's often identify the same resource, it is likely that different users could use different IDs for that resource. Using daml:equivalentTo, content from these two sources can be effectively merged.

The daml:versionInfo property allows an ontology to provide version information. However, in DAML the contents of this property are not defined, thus it cannot be used to automatically determine prior versions of the ontologies. Furthermore, there is no way to indicate a backward-compatible version. Therefore, compatible ontology perspectives (see Definition 3.31) cannot be used with DAML.

DAML also includes support for XML Schema data types. All data types are considered special classes and each has an identifier that is constructed from the URL of its source document and its name. Thus, a data value can be assigned a type, as in:

```
<xsd:decimal rdf:value="3.14" />
```

Also, the range of a property could be a data type:

```
<daml:DatatypeProperty rdf:ID="height">
  <rdfs:range rdf:resource=
        "http://www.w3.org/2000/10/XMLSchema#decimal"/>
</daml:DatatypeProperty>
```

Since DAML is based on description logics, it has all of the advantages of OIL. However, since it is also based on RDF, it has many of the disadvantages of that language. Despite all of its features, DAML is still not more expressive than SHOE. Although the cardinality constraints and boolean expressions can express things that cannot be expressed in SHOE, SHOE's Horn clause-like inference rules can express things not possible in DAML. However, there are plans for a DAML-Logic language, that will extend DAML+OIL with some fragment of first-order logic. Such a language would most likely be more expressive than SHOE.

7.5 Summary

We have described four different semantic web languages, and compared them to SHOE. In Table 7.1, we summarize the results of this comparison. In this table, syntax describes the notation of the language. Class hierarchy indicates whether the language allows taxonomies of classes to be defined. The Horn logic, description logic, and predicate logic rows all indicate the ability of a language to express the axioms of that kind of logic. Class and predicate equivalence deals with the language's ability to establish equivalence between classes and predicates (known as relations or properties in some languages), respectively. Instance equivalence concerns the ability to express the equivalence of individuals. A language that has decentralized ontologies allows ontologies to be developed autonomously, and does not require a central authority to approve or control them. Ontology extension is the

Feature	SHOE	Ontobroker	RDFS	OIL	DAML+OIL
Syntax	SGML/XML	HTML	XML	RDF	RDF
Formal semantics	Yes	Yes	No	Yes	Yes
Class hierarchy	Yes	Yes	Yes	Yes	Yes
Horn logic	Yes	Yes	No	No	No
Description logic	No	No	No	Yes	Yes
Predicate logic	No	Yes	No	No	No
Class equivalence	Yes	Yes	No	No	Yes
Predicate equivalence	Yes	Yes	No	No	Yes
Instance equivalence	No	No	No	No	Yes
Decentralized ontologies	Yes	No	Yes	Yes	Yes
Ontology extension	Yes	No	Yes*	Yes*	Yes
Ontology revision	Yes	No	No	No	No
Revision compatibility	Yes	No	No	No	No

Table 7.1: Comparison of semantic web languages.

ability of the language to include other ontologies. RDFS and OIL are marked with an asterisk because this feature is not explicit, but can be implied by namespaces. Ontology revision concerns the ability to explicitly state that one ontology is a new version of another ontology, and a language allows revision compatibility if it can state that a revision is backward-compatible with specific prior versions.

Chapter 8

Conclusion

In this chapter, we will review the SHOE language and systems, and discuss how they meet the needs of the Semantic Web. Based on this analysis, we will recommend future directions for semantic web research. We then speculate as to how the Semantic Web could revolutionize the way people use the Internet.

8.1 Analysis

In this thesis, we have described the challenges that must be overcome to realize the Semantic Web. The first problem is to extract structured knowledge from web pages. Then we must be able to integrate the data that is extracted from different sources. We have shown how ontologies can be used for integration, but since the Web is a distributed system with information on a multitude of sources, a single ontology solution is unrealistic. To accommodate the needs of diverse information providers, we allow ontologies to be created autonomously, but provide means for extending other ontologies to enable integration at design time. We pointed out that ontologies on the Web will need to evolve, and included features to maximize integration between data that commits to different versions of ontologies. A final problem is creating systems that can scale to the Web's enormous size. We will now examine our proposed solutions to these challenges.

In order to enable any agent to easily extract content from web pages, we can use SHOE tags to formally describe the knowledge. Although, this is one of SHOE's greatest strengths, it can also be its Achilles heel. There are already a billion web pages in existence, and convincing their owners to add semantic tags will not be an easy task. A page author will only add the tags if they can help lead others to the web page, but users will not query SHOE systems unless there is sufficient SHOE content to provide a likely match. In his description of the early days of developing the Web, Tim Berners-Lee describes a similar problem: [5, page 30]

> A big incentive for putting a document on the Web was that anyone else in the world could find it. But who would bother to install a client if there wasn't exciting information already on the Web? Getting out of this chicken-and-egg situation was the task before us.

In Chapters 5 and 6, we presented tools and techniques that reduce the level of effort needed to create SHOE markup. We also presented tools that demonstrate how the markup can be used. We believe that tools like these will demonstrate that the benefit of using a semantic web language outweighs the cost of of annotating pages. Still, there is the chicken-and-egg problem, and large quantities of useful semantic web content must be created to get the ball rolling. This is already beginning to emerge, due to projects like DAML at DARPA.

In Chapter 6, we described how SHOE could be used in two different domains. In the process, we found that certain types of pages were more amenable to annotation than others. In particular, pages with many hyperlinks that provided succinct information in labeled field, list, or table formats seem to be the best kinds of pages for this process. In general, people had difficulty identifying relevant instances and relations from long prose documents with few hyperlinks. Any effort to use SHOE must take into account the content and organization of the source web pages, so that an appropriate strategy for markup can be developed.

A significant issue for the Semantic Web is establishing the identity of individuals. The approach taken by SHOE is to choose a unique URL for each individual. In many cases, such as when the individual is a person or organization with a homepage, this is an acceptable solution. However, some people have multiple homepages (such as a personal page and professional page), and the URLs of a person can change over time (if the person changes jobs or internet service providers). Furthermore, problems arise with objects that are not owned by any person or organization. For example, who has the right to choose keys for Napoleon, the continent of Asia, the 16th century, or the Sun? In SHOE, we would create constants in ontologies to describe each of these concepts, but this begs the question of who has the right to create these ontologies. Perhaps the answer lies not in choosing a particular identifier for a concept, but in allowing a page to assign any unique identifier it chooses and state that it is equivalent to other identifiers for the same concept. DAML's equivalentTo feature provides just this capability, and could be one of the most important features of a Semantic Web language.

A fundamental component of the SHOE approach is that every document must commit to a formally described ontology. This provides a compromise between the extremes of a single, universal schema for the entire Web and individual schemas for each document. However, unlike XML DTDs, SHOE ontologies are arranged in a taxonomy with generic ontologies at the top that are extended by domain-specific ontologies. This approach encourages reuse while simultaneously allowing arbitrary extension. Ontologies can serve as contexts where different sets of axioms are applicable. Furthermore, ontology axioms can be used to articulate between the many common types of representational differences, providing a declarative means for transformation. However, SHOE lacks features such as arithmetic functions, aggregation, and string manipulation functions that are needed to provide articulation axioms for the full range of representational differences.

Due to the Web's dynamic nature, semantic web ontologies will inevitably need to change, but these changes could adversely affect the various resources that commit to the ontologies. In SHOE, each revision of an ontology is a separate resource with its own unique version number, and instances commit to a specific version of an ontology. This ensures that dependencies are not broken. Furthermore, the definition of compatible ontology perspectives allows SHOE to integrate data from resources that committed to different versions of an ontology. This approach seems to work well, even in the presence of major structural changes to the ontology such as those described in Section 6.1.5.

To deal with the quantity of information on the Semantic Web, the language must be scalable. Although we limited SHOE to the expressivity of datalog, this may still not scale to the needs of all applications. Therefore, we indicated that SHOE could use a number of different knowledge representation systems, which differed in the completeness of reasoning and query response time. For example, a deductive database can support the full semantics of SHOE, but there will be performance tradeoffs. On the other hand, relational database management systems can provide performance gains but at the cost of no inference.

We believe that SHOE has demonstrated many of the features needed in a semantic web language. SHOE provides interoperability in distributed environments through the use of extensible, shared ontologies, the avoidance of contradictions, and localization of inference rules. It handles the changing nature of the Web with an ontology versioning scheme that supports backward-compatibility. It takes steps in the direction of scalability by limiting expressivity and allowing for different levels on inferential support. Finally, since the Web is an "open-world," SHOE does not allow conclusions to be drawn from lack of information.

8.2 Future Directions

Although we believe SHOE is good language that has practical use, we do not mean to suggest that it solves all of the problems of the Semantic Web. We are at the beginning of a new and exciting research field and there is still much work to do. In this section, we discuss some possibilities for future work.

8.2.1 User Friendly Tools

Clearly, since the Semantic Web is intended to be an extension of the contemporary Web, its tools must be usable by the layperson. Since query tools will be used most often, the greatest attention should be paid them. Ideally, we want a query tool that is as easy to use as as keyword-based search engine, but provides the accurate answers possible with structured queries. The first challenge for such a tool is establishing the context of the query. Although it is easy to choose a query context from the pick list in our SHOE Search tool when there are only a dozen ontologies to choose from, the Semantic Web may have thousands of ontologies. How can a user choose the right context from this set? Another problem is how can a user learn enough about an ontology's contents to create the desired query. Most users want instant answers, and will not bother to peruse pages of ontology documentation just so they can form a query. Perhaps the answer is to allow the user to type a query string, which the system attempts to parse into a structured query, prompting the user to disambiguate terms when necessary.

Another important aspect of usability is the creation of SHOE documents. SHOE hopes to overcome the knowledge acquisition bottleneck by amortizing the cost of annotation over the entire set of content providers, but unless the process is straightforward, this is doomed to failure. We are actively working with our users to determine what interfaces are the most intuitive. Certainly, the ultimate annotation process would be fully automatic, but due to limitations of NLP in general domains, this goal is currently unrealistic. However, a semi-automatic method that incorporated ideas from NLP or machine learning may simplify the process for the user.

8.2.2 Scalability

Because the scope of the Semantic Web is as broad as that of the contemporary Web, scalability is critical to any Semantic Web system. We intentionally restricted SHOE to the expressivity of datalog so that we could use reasoning algorithms developed for large data sets. However, it is unclear if even this restriction will truly allow us to scale to problems with thousands of ontologies and billions of assertions as will be required on the Semantic Web. In Section 5.1.4, we discussed how different knowledge base systems could provide different inferential capabilities and performance characteristics, and suggested that incomplete reasoners would provide better performance. Future experiments need to verify this hypothesis and explore the tradeoffs, so that well-informed decisions can be made.

We conducted a preliminary experiment that compared the use of XSB and Parka as SHOE repositories in the computer science domain. We chose ten representative queries, which ranged from one to five conjuncts and had up to four variables. Seven of these queries included a predicate which was partially defined by one of the ontology's inference rules. Two of the predicates used partial string matching, which can help users locate instances without knowing their complete names. Using a Sparc 20 with 128 megabytes of memory, we issued the queries to both KBs, and measured the system's response time (in milliseconds) and the number of answers returned for each query.

The results of the experiment are shown in Table 8.1. Response times for XSB varied from 39ms to 124215ms, while the response times for Parka varied from 376ms to 2991ms. Although, XSB out-performed Parka on half of the queries, this can be partially attributed to the different ways the SHOE KB library accesses the repositories. XSB is executed as a child process, whose input and output streams are managed by the SHOE software. Parka, on the other hand, is a client-server application, and the SHOE KB library communicates with it via sockets. Thus, a dominant cost in the Parka timings was the additional overhead of socket communication, which can be as much as 2000 ms. An interesting feature of the XSB timings is that three queries took over 10 seconds to complete. The one thing in common between these queries is that each involved a test for membership in a high-level category. Due to child categories and to relations that had arguments typed to the high-level categories, the corresponding predicates of these categories appear in the heads of many rules. This caused XSB's search to branch out significantly more than in other queries and resulted in the longer completion times. In general, it seems that the smaller variance in query completion time makes Parka a better choice for queries where response time is important.

Query	Time (ms)		No. of Answers	
	XSB	Parka	XSB	Parka
$instanceOf(x, University)$	2248	2430	27	27
$instanceOf(x, Organization)$	13031	1337	254	254
$member(http://www.cs.umd.edu/, x)$	1040	1663	480	478
$member(http://www.umd.edu/, x)$	1121	376	480	0
$member(http://www.cs.umd.edu/, x) \wedge instanceOf(x, Faculty)$	1373	1010	57	57
$member(x, http://www.cs.umd.edu/users/heflin/)$	39	1016	5	4
$instanceOf(x, Department) \wedge member(x, http://www.cs.umd.edu/users/heflin/)$	72	979	1	1
$instanceOf(p, Publication) \wedge publicationAuthor(p, a) \wedge member(http://www.cs.stanford.edu/, a)$	20208	2091	23	23
$instanceOf(p, Article) \wedge publicationAuthor(p, a) \wedge name(a, n) \wedge stringMatch(n,$ "Heflin" $)$	352	2991	7	7
$instanceOf(x, Person) \wedge publicationAuthor(p, x) \wedge member(o, x) \wedge name(o, n) \wedge stringMatch(n,$ "Stanford" $)$	124215	2092	46	23

Table 8.1: Comparison of XSB and Parka.

The other aspect of the experiment was the degree of completeness in the returned answers. While XSB always provided complete answers, Parka only provided complete answers for 6 of the 10 queries. In one case (when querying the members of the UMCP CS department), Parka could not provide any answers, although XSB returned 480. This was because no page explicitly contained assertions about members of *http://www.umd.edu/*, but XSB was able to use the rule that stated membership transfers through the subOrganization relation to infer that all members of *http://www.cs.umd.edu/* were also members of *http://www.umd.edu/*. In another query, which concerned the people who authored publications and were members of an organization with Stanford in its name, XSB returned twice as many answers as Parka. The reason is that XSB returned each person twice, once each for being a member of "Stanford University" and the "Stanford University Computer Science Department." Since Parka was unable to determine that the individuals were members of the university, it only returned them as members of the computer science department. In the other two queries, Parka only returned one or two results less than XSB. Although, Parka's lack of a complete inference engine for SHOE only seriously affected one query in this set, larger and richer ontologies are likely to have more inference rules, and the differences between Parka and XSB on queries using these ontologies will be much more significant.

This comparison indicates that Parka's query response times are much more reliable than XSB's, where certain queries can result in an intolerable delay. However, there are certain queries for which Parka is useless. It is clear then that each system is better in certain situations. Obviously, a more thorough experiment is needed. Such an experiment should ensure that additional variables such as network latency are accounted for, test a wider range of systems, including relational databases; compare variations in the way SHOE is implemented in each system, and use larger and more realistic knowledge bases for the tests.

8.2.3 Language Design

Although SHOE was the first ontology-based web language, there are many directions for possible improvement. Researchers need to develop a set of criteria for evaluating Semantic Web languages. Based on our analysis in Chapter 3, we suggest a few basic requirements. First, the language must be able to define ontologies, and ontologies must be able to extend and revise other ontologies. Second, the language must have the power to express translations between different representations of the same concepts, and particularly include the ability to establish equivalence of terms. Third, the language must have an XML syntax, so that it can make use of existing infrastructure.

In order to translate between different representations of the same concepts, a language would need additional primitives not present in SHOE. For example, in order to convert between different measurement units, the language must allow arithmetic functions in inference rules. However, if an arithmetic function is used recursively in a rule, inference procedures may never terminate. Aggregation and string manipulation are needed for other types of conversions. To provide truly flexible facilities, arbitrary functions should be considered, but since the definition of an arbitrary function would require a much more complex language, it should be specified by means of a remote procedure call.

Although, SHOE takes an open-world approach, there are many useful queries and actions that cannot be performed by web agents without closed-world information. Localized closed-world (LCW) statements [26] are a promising step in this direction. LCW statements can be used to state that the given source has all of the information on a given topic. LCW statements are more appropriate for the Web than the closed-world assumption, but there is still a question as to how a query system acquires the set of LCW statements that could be relevant. One possible extension to SHOE is to allow LCW statements to be expressed in the language.

Finally, research must be conducted to establish the best set of primitives for a semantic web language. For example, the limited expressivity of SHOE did lead to occasions where incorrect usage of an ontology's vocabulary could not be detected, resulting in erroneous conclusions. A language that could express negation, disjoint sets, and/or cardinality constraints could use those features to validate and evaluate data that is discovered. However, in a distributed system the violation of a constraint may be due to bad data that was discovered earlier. As such, a constraint should not prevent data from being included in the KB, but should instead be used as a filter at query time that results in a warning or lowering of the confidence in a particular assertion. Features from other logics may also be useful. For example, temporal, probabilistic, or fuzzy logics might have a place on the Semantic Web.

Clearly, there are a number of features that could be of use in a semantic web language. To properly evaluate candidate languages, we need to identify what expressive needs are most important and compare complexity of the languages. An important direction for future work is an enumeration of the possible features along with an analysis of the cost, complexity and benefit of each feature.

8.2.4 Web Services and Agent Communication

In this thesis, we focused on the Semantic Web as a search and query mechanism, but it can be much more. The Web already provides a number of services, some of which look up information (such as flight schedules) and some of which perform an action (such as book a flight). However if these services could be described with semantic markup, then intelligent agents can use these services. Preliminary research in this area is described by McIlraith, Son, and Zeng [71]. A crucial problem to be solved here is the design of an ontology that is flexible enough to describe the wide range of potential services.

Web services can be thought of as simple agents, and techniques for describing them could be expanded for use in agent communication [53]. An agent can advertise its capabilities using a service ontology. Other agents could then understand this advertisement and determine if they should request a service from the advertising agent. Ontologies may also be needed for the process of negotiation and for rating of agent services.

8.2.5 A Web of Trust

A serious problem on the contemporary Web is finding reliable information. This problem will be even more crucial for the Semantic Web, where agents will be integrating information from multiple sources. If an incorrect premise is used due to a single faulty source, then any conclusions drawn may be in error. In the perspective approaches of Chapter 3, it is assumed that all resources used to form a perspective are reliable. This approach could be extended with an additional trust function which extracts from the set of resources only those assertions that are deemed reliable in some way.

One problem with trust is that it can be very subjective, and two individuals may disagree on whether a particular source is reliable. To further complicate matters, a given source may be reliable only on certain subject matter and reliability may depend on supporting evidence. A potential solution to these problems is to create special ontologies that provide belief systems using sets of rules that filter out claims which may be suspect. Such ontologies will require special relations such as $claims(x, a)$, which is true if x is the source of assertion a, and $believe(a)$, which is true if the agent should believe assertion a [50]. Users can then subscribe to a particular belief system or customize one to their own needs.

8.3 A Vision of the Semantic Web

The Semantic Web, which was was once the dream of a few isolated individuals, is now on the verge of revolutionizing the Internet. The DAML project has helped to garner the cause widespread attention, and the W3C's Semantic Web activity is well underway. In the summer of 2001, a working group will form to develop a W3C standard for web ontologies. By 2003, this standard should be in place and will serve as the foundation of a Semantic Web.

Then, the Internet will begin to be transformed in amazing ways. First, people will be able to conduct more accurate searches, and the answers they receive may be based on the automatic integration of numerous sources. As with the current Web, there will be many search engines to choose from, but these engines will differ not just in coverage of the Web, but also in inferential completeness and query response time. Some search engines will have a single trust model that represents one viewpoint of the Web, while others may allow users to configure their own trust model. However, since the details of many web pages will be too complex to annotate, there will still be a place for keyword-based search engines on the Web. The best search engines will combine keyword and semantic web search techniques to best satisfy their users.

Although improved search is a significant capability of the Semantic Web, the real revolution will occur with agents that don't just *find* things, but also *do* things. These agents will be automated personal secretaries that interact with each other over the Internet. For example, you could tell your agent to make travel arrangements for you to attend a conference. The agent would go the conference web page to find out about the location and dates of the conference. It would then use your personal information and preferences to determine a means of transportation, contact a travel site, and make plane and hotel reservations. The agent may also discover that your favorite band is playing in town during your stay, and even though the show is sold out, the agent might find reasonably priced tickets from a ticket broker. In addition to presenting your itinerary, the agent would mention the concert and ask if you to wanted to purchase the tickets. Although this may sound far-fetched, when web pages are annotated in a semantic markup language, the problem becomes much easier to solve.

Semantic web agents will also bring changes to e-commerce. People will be able to search for products that contain exactly the features they desire and at the lowest possible price. If the seller also has an agent, then the two agents may even negotiate a lower price for you. The impact of this on the Web will be that sellers who wish to stay in business will have to constantly match their competitors prices or differentiate themselves on some other aspect such as features of the product, quality, or customer service. All of these things can be described on the Semantic Web and could play in the consideration of a product for the agent.

Appendix A

SHOE DTDs

This appendix contains the DTDs for both the SGML and XML syntaxes of SHOE. These DTDs specify what tags can be used in a document, their structure, and how they may be nested.

A.1 SGML DTD

The SGML syntax of SHOE is an application of the Standard Generalized Markup Language (SGML). An SGML DTD describes a document structure using element, attribute, and entity declarations. Comments are indicated with <!- - and - ->.

An element declaration consists of <!ELEMENT, the element name, minimization options, a content specification, and a >. The minimization options consists of two characters indicating the minimization for the start and end tags of the element. The character - indicates the tag is required, while O indicates that it is optional. The content specification can be declared or a model group. In this DTD, the only declared content is CDATA, which stands for character data and means the content is text that does not include any tags. Model groups are used to specify the subtags of an element. Here, "," separates elements that must appear in sequence, "|" separates items of a choice, and "&" separates items that must appear in any order. The quantity of elements is indicated with "?" for optional but cannot repeat, "*" for optional but can repeat, and "+" for mandatory and repeatable.

Attribute declarations associate attributes with elements and have the form <!ATTLIST, element name, a sequence of attribute definitions, and a >. Each attribute definition consists of an attribute name, declared value, and default value. The only declared values used here are CDATA or lists of literal values. The default value can be #REQUIRED indicating that the attribute must appear in every tag, #IMPLIED indicating that it may be absent, or a specific value.

Finally, entities are like macros, in that they can associate a name with some component of the DTD. An entity declaration consists of <!ENTITY %, the entity name, an optional system identifier, some content, and a >. An entity's content is inserted wherever the entity is referenced with %entityname;. A external entity can be referenced by using the PUBLIC system identifier and specifying the name of the public entity.

The DTD below builds on the HTML 3.2 DTD by redefining the block entity to include the elements ONTOLOGY and INSTANCE, and then defining the corresponding sub-elements. It then includes the HTML DTD as a public entity.

```
<!-- DTD for SHOE -->
<!-- Last Mod: 1/1/98 -->

<!ENTITY % shoe.content "ONTOLOGY | INSTANCE" >

<!-- The following three entity declarations are used to
     override the HTML content model for blocks, so that
     an ONTOLOGY or INSTANCE can appear anywhere a block
     can. Typically this is as a top level element in the
     BODY of the HTML document -->

<!ENTITY % list "UL | OL |  DIR | MENU">
```

```
<!ENTITY % preformatted "PRE">

<!ENTITY % block
     "P | %list | %preformatted | DL | DIV | CENTER |
      BLOCKQUOTE | FORM | ISINDEX | HR | TABLE |
      %shoe.content;">


<!-- Declarations for ontologies -->
<!ELEMENT ONTOLOGY   - -  (USE-ONTOLOGY | DEF-CATEGORY |
                           DEF-RELATION | DEF-RENAME |
                           DEF-INFERENCE | DEF-CONSTANT |
                           DEF-TYPE)* >

<!ATTLIST ONTOLOGY
        id                 CDATA        #REQUIRED
        version            CDATA        #REQUIRED
        description        CDATA        #IMPLIED
        declarators        CDATA        #IMPLIED
        backward-compatible-with    CDATA     #IMPLIED >

<!ELEMENT USE-ONTOLOGY    - O  EMPTY >
<!ATTLIST USE-ONTOLOGY
        id                 CDATA        #REQUIRED
        version            CDATA        #REQUIRED
        prefix             CDATA        #REQUIRED
        url                CDATA        #IMPLIED >

<!ELEMENT DEF-CATEGORY    - O  EMPTY >
<!ATTLIST DEF-CATEGORY
        name               CDATA        #REQUIRED
        isa                CDATA        #IMPLIED
        description        CDATA        #IMPLIED
        short              CDATA        #IMPLIED >

<!ELEMENT DEF-RELATION    - -  (DEF-ARG)* >
<!ATTLIST DEF-RELATION
        name               CDATA        #REQUIRED
        short              CDATA        #IMPLIED
        description        CDATA        #IMPLIED >

<!ELEMENT DEF-ARG         - O  EMPTY >
<!ATTLIST DEF-ARG
        pos                CDATA        #REQUIRED
        type               CDATA        #REQUIRED
```

```
        short              CDATA        #IMPLIED >
<!-- pos must be either an integer, or one of the
     strings: FROM or TO -->

<!ELEMENT DEF-RENAME      - O  EMPTY >
<!ATTLIST DEF-RENAME
        from               CDATA        #REQUIRED
        to                 CDATA        #REQUIRED >

<!ELEMENT DEF-CONSTANT    - O  EMPTY >
<!ATTLIST DEF-CONSTANT
        name              CDATA        #REQUIRED
        category          CDATA        #IMPLIED >

<!ELEMENT DEF-TYPE        - O  EMPTY >
<!ATTLIST DEF-TYPE
        name               CDATA        #REQUIRED
        description        CDATA        #IMPLIED
        short              CDATA        #IMPLIED >

<!-- Declarations for inferences -->
<!-- Inferences consist of if and then parts, each of
     which can contain multiple relation and category
     clauses -->
<!ELEMENT DEF-INFERENCE   - - (INF-IF, INF-THEN) >
<!ATTLIST DEF-INFERENCE
        description        CDATA         #IMPLIED >
<!ELEMENT INF-IF          - -  (CATEGORY | RELATION |
                                COMPARISON)+ >
<!ELEMENT INF-THEN        - -  (CATEGORY | RELATION)+ >
<!ELEMENT COMPARISON      - -  (ARG, ARG) >
<!ATTLIST COMPARISON
        op        (equal | notEqual | greaterThan |
                   greaterThanOrEqual | lessThanOrEqual |
                   lessThan)        #REQUIRED >

<!-- Declarations for instances -->
<!ELEMENT INSTANCE        - -  (USE-ONTOLOGY | CATEGORY |
                                RELATION | INSTANCE)* >
<!ATTLIST INSTANCE
        key                CDATA        #REQUIRED
        delegate-to        CDATA        #IMPLIED >

<!ELEMENT CATEGORY        - O  EMPTY >
<!ATTLIST CATEGORY
```

```
        name              CDATA        #REQUIRED
        for               CDATA        #IMPLIED
        usage             (VAR | CONST)        CONST >
<!-- If VAR is specified for a category that is not
     within a <DEF-INFERENCE>, then it is ignored -->

<!ELEMENT RELATION        - -  (ARG)* >
<!ATTLIST RELATION
        name              CDATA        #REQUIRED >
<!ELEMENT ARG             - O  EMPTY >
<!ATTLIST ARG
        pos               CDATA        #REQUIRED
        value             CDATA        #REQUIRED
        usage             (VAR | CONST)        CONST >

<!-- pos must be either an integer, or one of the
     strings: FROM or TO.  -->
<!-- If VAR is specified for an arg that is not within a
     <DEF-INFERENCE>, then it is ignored -->

<!-- Include DTD for HTML -->
<!ENTITY % HTMLDTD PUBLIC
           "-//W3C//DTD HTML 3.2 Final//EN" >
%HTMLDTD;
```

A.2 XML DTD

Since XML is essentially a simplified version of XML, the XML syntax for SHOE is very similar to the SGML one. Likewise, the XML DTD is very similar. The key differences between the two DTDs are:

- XML is case-sensitive, so the case of all element and attribute names is relevant. We chose lower-case to correspond with XHTML.
- XML does not allow tag minimization, so the tag minimization tokens are not used in element declarations.
- This DTD can stand on its own, and does not need to reference the HTML DTD.

```
<!ELEMENT shoe              (ontology | instance)* >

<!-- Since this may be embedded in a document that
     doesn't have META elements, the SHOE version number
     is included as an attribute of the shoe element. -->
<!ATTLIST shoe
        version            CDATA        #REQUIRED >

<!-- Declarations for ontologies -->
<!ELEMENT ontology          (use-ontology | def-category |
                             def-relation | def-rename |
                             def-inference | def-constant |
                             def-type)* >

<!ATTLIST ontology
        id                 CDATA        #REQUIRED
        version            CDATA        #REQUIRED
        description        CDATA        #IMPLIED
        declarators        CDATA        #IMPLIED
        backward-compatible-with    CDATA     #IMPLIED >

<!ELEMENT use-ontology     EMPTY >
<!ATTLIST use-ontology
        id                 CDATA        #REQUIRED
        version            CDATA        #REQUIRED
        prefix             CDATA        #REQUIRED
        url                CDATA        #IMPLIED >

<!ELEMENT def-category     EMPTY >
<!ATTLIST def-category
        name               CDATA        #REQUIRED
        isa                CDATA        #IMPLIED
        description        CDATA        #IMPLIED
        short              CDATA        #IMPLIED >

<!ELEMENT def-relation     (def-arg)* >
<!ATTLIST def-relation
        name               CDATA        #REQUIRED
        short              CDATA        #IMPLIED
        description        CDATA        #IMPLIED >

<!ELEMENT def-arg          EMPTY >
<!ATTLIST def-arg
        pos                CDATA        #REQUIRED
        type               CDATA        #REQUIRED
```

```
        short              CDATA        #IMPLIED >
<!-- pos must be either an integer, or one of the
     strings: FROM or TO -->

<!ELEMENT def-rename      EMPTY >
<!ATTLIST def-rename
        from               CDATA        #REQUIRED
        to                 CDATA        #REQUIRED >

<!ELEMENT def-constant    EMPTY >
<!ATTLIST def-constant
        name               CDATA        #REQUIRED
        category           CDATA        #IMPLIED >

<!ELEMENT def-type        EMPTY >
<!ATTLIST def-type
        name               CDATA        #REQUIRED
        description        CDATA        #IMPLIED
        short              CDATA        #IMPLIED >

<!-- Declarations for inferences -->
<!-- Inferences consist of if and then parts, each of
     which can contain multiple relation and category
     clauses -->
<!ELEMENT def-inference   (inf-if, inf-then) >
<!ATTLIST def-inference
        description        CDATA         #IMPLIED >
<!ELEMENT inf-if          (category | relation |
                           comparison)+ >
<!ELEMENT inf-then        (category | relation)+ >
<!ELEMENT comparison      (arg, arg) >
<!ATTLIST comparison
        op       (equal | notEqual | greaterThan |
                  greaterThanOrEqual | lessThanOrEqual |
                  lessThan)        #REQUIRED >

<!-- Declarations for instances -->
<!ELEMENT instance        (use-ontology | category |
                           relation | instance)* >
<!ATTLIST instance
        key                CDATA        #REQUIRED
        delegate-to        CDATA        #IMPLIED >

<!ELEMENT category        EMPTY >
<!ATTLIST category
```

```
        name              CDATA        #REQUIRED
        for               CDATA        #IMPLIED
        usage             (VAR | CONST)        "CONST" >
<!-- If VAR is specified for a category that is not
     within a <def-inference>, then it is ignored -->

<!ELEMENT relation        (arg)* >
<!ATTLIST relation
        name              CDATA        #REQUIRED >
<!ELEMENT arg             EMPTY >
<!ATTLIST arg
        pos               CDATA        #REQUIRED
        value             CDATA        #REQUIRED
        usage             (VAR | CONST)        "CONST" >

<!-- pos must be either an integer, or one of the
     strings: FROM or TO -->
<!-- If VAR is specified for an arg that is not within a
     <def-inference>, then it is ignored -->
```

BIBLIOGRAPHY

[1] G. Arocena, A. Mendelzon, and G. Mihaila. Applications of a web query language. In *Proceedings of ACM PODS Conference*, Tuscon, AZ, 1997.

[2] N. Ashish and C. Knoblock. Semi-automatic wrapper generation for internet information sources. In *Proceedings of the Second IFCIS Conference on Cooperative Information Systems (CoopIS)*, Charleston, SC, 1997.

[3] T. Berners-Lee and D. Connolly. *Hypertext Markup Language - 2.0*. IETF HTML Working Group, November 1995. At: http://www.ics.uci.edu/pub/ietf/html/rfc1866.txt.

[4] T. Berners-Lee, J. Hendler, and O. Lasilla. The Semantic Web. *Scientific American*, May 2001.

[5] Tim Berners-Lee. *Weaving the Web*. HarperCollins Publishers, New York, NY, 1999.

[6] P. Biron and A. Malhotra. *XML Schema Part 2: Datatypes*. W3C, May 2001. At: http://www.w3.org/TR/2001/REC-xmlschema-2-20010502.

[7] D. Bobrow and T. Winograd. An overview of KRL, a knowledge representation language. *Cognitive Science*, 1(1), 1977.

[8] Alexander Borgida. On the relationship between description logic and predicate logic. In *Proceedings of CIKM-94*, pages 219–225, 1994.

[9] A. Bouguettaya, B. Benatallah, and A. Elmagarmid. An overview of multidatabase systems: Past and present. In Elmagarmid et al. [25].

[10] R. Brachman. What IS-A is and isn't: An analysis of taxonomic links in semantic networks. *IEEE Computer*, 16(10):30–36, October 1983.

[11] R. Brachman and H. Levesque. The tractability of subsumption in frame-based description languages. In *Proc. of the National Conference on Artificial Intelligence (AAAI-1984)*, pages 34–37, Menlo Park, CA, 1984. AAAI/MIT Press.

[12] R. Brachman, D. McGuinness, P.F. Patel-Schneider, L. Resnick, and A. Borgida. Living with Classic when and how to use a KL-ONE-like language. In J. Sowa, editor, *Explorations in the Representation of Knowledge*. Morgan-Kaufmann, CA, 1991.

[13] R. Brachman and J. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2), 1985.

[14] T. Bray, D. Hollander, and A. Layman. *Namespaces in XML*. W3C, January 1999. At: http://www.w3.org/TR/1999/REC-xml-names-19990114/.

[15] T. Bray, J. Paoli, and C. Sperberg-McQueen. *Extensible Markup Language (XML)*. W3C (World Wide Web Consortium), February 1988. At: http://www.w3.org/TR/1998/REC-xml-19980210.html.

[16] D. Brickley and R.V. Guha. *Resource Description Framework (RDF) Schema Specification (Candidate Recommendation)*. W3C (World-Wide Web

Consortium), March 2000. At:
http://www.w3.org/TR/2000/CR-rdf-schema-20000327.

[17] V. Chaudhri, A. Farquhar, R. Fikes, P. Karp, and J. Rice. OKBC: A programatic foundation for knowledge base interoperability. In *Proc. of the Fifteenth National Conference on Artificial Intelligence (AAAI-1998)*, pages 600–607, Menlo Park, CA, 1998. AAAI/MIT Press.

[18] J. Clark. *XSL Transformations (XSLT)*. W3C (World-Wide Web Consortium), November 1999. At: http://www.w3.org/TR/1999/REC-xslt-19991116.

[19] C. Collet, M. Huhns, and W. Shen. Resource integration using a large knowledge base in Carnot. *IEEE Computer*, 24(12), December 1991.

[20] M. Craven, D. DiPasquo, D. Freitag, A. McCallum, T. Mitchell, K. Nigram, and S. Slattery. Learning to extract symbolic knowledge from the World Wide Web. In *Proc. of the Fifteenth National Conference on Artificial Intelligence (AAAI-1998)*, Menlo Park, CA, 1998. AAAI/MIT Press.

[21] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

[22] S. Decker, D. Fensel, F. van Harmelen, I. Horrocks, S. Melnik, M. Klein, and J. Broekstra. Knowledge representation on the Web. In *Proceedings of the 2000 International Workshop on Description Logics (DL2000)*, Aachen, Germany, August 2000.

[23] Stefan Decker, Michael Erdmann, Dieter Fensel, and Rudi Studer. Ontobroker: Ontology based access to distributed and semi-structured information. In

Semantic Issues in Multimedia Systems. Proceedings of DS-8, pages 351–369, Boston, 1999. Kluwer Academic Publisher.

[24] S. Dobson and V. Burrill. Lightweight databases. *Computer Networks and ISDN Systems*, 27(6):1009–1015, 1995.

[25] A. Elmagarmid, M. Rusinkiewicz, and A. Sheth, editors. *Management of Heterogeneous and Autonomous Database Systems*. Morgan Kaufmann, San Francisco, 1999.

[26] O. Etzioni, K. Golden, and D. Weld. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence*, 89:113–148, 1997.

[27] M. Evett, W. Andersen, and J. Hendler. Providing computational effective knowledge representation via massive parallelism. In L. Kanal, V. Kumar, H. Kitano, and C. Suttner, editors, *Parallel Processing for Artificial Intelligence*. Elsevier Science, Amsterdam, 1993.

[28] A. Farquhar, A. Dappert, R. Fikes, and W. Pratt. Integrating information sources using context logic. In *Proceedings of AAAI Spring Symposium on Information Gathering from Distributed, Heterogeneous Environments*, 1995. Also available as KSL Technical Report KSL-95-12.

[29] A. Farquhar, R. Fikes, and J. Rice. The Ontolingua Server: A tool for collaborative ontology construction. *International Journal of Human-Computer Studies*, 46(6):707–727, 1997.

[30] Adam Farquhar, Richard Fikes, and James Rice. Tools for assembling modular ontologies in Ontolingua. In *Proc. of Fourteenth American Association for*

Artificial Intelligence Conference (AAAI-97), pages 436–441, Menlo Park, CA, 1997. AAAI/MIT Press.

[31] D. Fensel, S. Decker, M. Erdmann, and R. Studer. Ontobroker: How to enable intelligent access to the WWW. In *AI and Information Integration, Papers from the 1998 Workshop*, Menlo Park, CA, 1998. AAAI Press. Technical Report WS-98-14.

[32] D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a nutshell. In *Knowledge Acquisition, Modeling, and Management, Proceedings of the European Knowledge Acquisition Conference (EKAW-2000)*, pages 1–16, Berlin, 2000. Springer-Verlag.

[33] D. Fensel, F. van Harmelen, I. Horrocks, D. McGuinness, and P. Patel-Schneider. OIL: An ontology infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[34] N. Foo. Ontology revision. In *Conceptual Structures: Third International Conference*, pages 16–31, Berlin, 1995. Springer-Verlag.

[35] ISO (International Organization for Standardization). ISO 8879:1986(E). information processing – text and office systems – Standard Generalized Markup Language (SGML), 1986.

[36] D. Freitag. Information extraction from HTML: Application of a general machine learning approach. In *Proc. of Fifteenth American Association for Artificial Intelligence Conference (AAAI-98)*, pages 517–523, Menlo Park, CA, 1998. AAAI/MIT Press.

[37] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[38] Peter Gardenförs. *Knowledge in Flux: Modeling the Dynamics of Epistemic States*. MIT Press, Cambridge, Mass., 1988.

[39] Michael Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA, 1987.

[40] Michael R. Genesereth, Arthur M. Keller, and Oliver Duschka. Infomaster: An information integration system. In *Proceedings of 1997 ACM SIGMOD Conference*, May 1997.

[41] M.R. Genesereth and R.E. Fikes. Knowledge Interchange Format, version 3.0 reference manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992.

[42] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[43] M. Grüninger. Designing and evaluating generic ontologies. In *Proceedings of ECAI'96 Workshop on Ontological Engineering*, 1996.

[44] N. Guarino. Formal ontology and information systems. In *Proceedings of Formal Ontology and Information Systems*, Trento, Italy, June 1998. IOS Press.

[45] N. Guarino and P. Giaretta. Ontologies and knowledge bases: Towards a terminological clarification. In N. Mars, editor, *Towards Very Large Knowledge*

Bases: Knowledge Building and Knowledge Sharing, pages 25–32. IOS Press, Amsterdam, 1995.

[46] R. V. Guha. *Contexts: A Formalization and Some Applications*. PhD thesis, Stanford University, 1991.

[47] J. Heflin and J. Hendler. Searching the Web with SHOE. In *Artificial Intelligence for Web Search. Papers from the AAAI Workshop*, pages 35–40, Menlo Park, CA, 2000. AAAI Press. Technical Report WS-00-01.

[48] J. Heflin and J. Hendler. Semantic interoperability on the Web. In *Proc. of Extreme Markup Languages 2000*, pages 111–120, Alexandria, VA, 2000. Graphic Communications Association.

[49] J. Heflin and J. Hendler. A portrait of the Semantic Web in action. *IEEE Intelligent Systems*, 16(2):54–59, 2001.

[50] J. Heflin, J. Hendler, and S. Luke. Reading between the lines: Using SHOE to discover implicit knowledge from the Web. In *AI and Information Integration, Papers from the 1998 Workshop*, Menlo Park, CA, 1998. AAAI Press. Technical Report WS-98-14.

[51] J. Heflin, J. Hendler, and S. Luke. Applying ontology to the Web: A case study. In J. Mira and J. Sanchez-Andres, editors, *International Work-Conference on Artificial and Natural Neural Networks, IWANN'99, Proceeding, Vol. II*, pages 715–724, Berlin, 1999. Springer.

[52] Jeff Heflin and James Hendler. Dynamic ontologies on the Web. In *Proc. of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 443–449, Menlo Park, CA, 2000. AAAI/MIT Press.

[53] J. Hendler. Agents and the Semantic Web. *IEEE Intelligent Systems*, 16(2):30–37, 2001.

[54] James Hendler and Deborah L. McGuinness. The DARPA agent markup language. *IEEE Intelligent Systems*, 15(6):72–73, November/December 2000.

[55] Joint US/EU Ad Hoc Agent Markup Language Committee. *DAML+OIL*, 2001. At: http://www.daml.org/2001/03/daml+oil-index.

[56] V. Kashyap and A. Sheth. Semantic similarities between objects in multiple databases. In Elmagarmid et al. [25].

[57] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42, 1995.

[58] Craig A. Knoblock, Steven Minton, Jose Luis Ambite, Naveen Ashish, Pragnesh Jay Modi, Ion Muslea, Andrew G. Philpot, and Sheila Tejada. Modeling web sources for information integration. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 211–218, Menlo Park, CA, 1998. AAAI/MIT Press.

[59] D. Konopnicki and O. Shemueli. W3QS: A query system for the World Wide Web. In *Proceedings of the 21st International Conference on Very Large Databases*, Zurich, Switzerland, 1995.

[60] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proc. of Fifteenth International Joint Conference on Artificial Intelligence*, pages 729–735, San Francisco, 1997. Morgan Kaufmann.

[61] O. Lassila. Web metadata: A matter of semantics. *IEEE Internet Computing*, 2(4):30–37, 1998.

[62] O. Lassila and R. Swick. *Resource Description Framework (RDF) Model and Syntax Specification*. World Wide Web Consortium (W3C), February 1999. At: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.

[63] D. Lenat, R.V. Guha, K. Pittman, D. Pratt, and M. Shepherd. Cyc: Toward programs with common sense. *Communications of the ACM*, 33(8):30–49, 1990.

[64] Douglas Lenat and R.V. Guha. *Building Large Knowledge-Based Systems*. Addison-Wesley, Reading, Mass., 1990.

[65] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987.

[66] S. Luke, L. Spector, D. Rager, and J. Hendler. Ontology-based web agents. In *Proceedings of the First International Conference on Autonomous Agents*, pages 59–66, New York, NY, 1997. Association of Computing Machinery.

[67] R. MacGregor. The evolving technology of classification-based knowledge representation systems. In J. Sowa, editor, *Explorations in the Representation of Knowledge*. Morgan-Kaufmann, CA, 1991.

[68] J. McCarthy. Notes on formalizing context. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence (IJCAI-93)*, pages 555–560, Los Altos, 1993. Morgan Kaufmann.

[69] Deborah L. McGuinness, Richard Fikes, James Rice, and Steve Wilder. An environment for merging and testing large ontologies. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000)*, Breckenridge, Colorado, April 2000.

[70] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[71] S. McIlraith, T. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[72] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, Los Altos, CA, 1988.

[73] M. Minsky. A framework for representing knowledge. In Patrick-Henry Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, New York, 1975.

[74] N. Noy and C. Hafner. The state of the art in ontology design. *AI Magazine*, 18(3):53–74, 1997.

[75] N. F. Noy and M. A. Musen. PROMPT: Algorithm and tool for automated ontology merging and alignment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, 2000.

[76] N. F. Noy, M. Sintek, S. Decker, M. Crubézy, R. W. Fergerson, and M. A. Musen. Creating semantic web contents with Protégé-2000. *IEEE Intelligent Systems*, 16(2):60–71, 2001.

[77] D. E. Oliver, Y. Shahar, M. A. Musen, and E. H. Shortliffe. Representation of change in controlled medical terminologies. *Artificial Intelligence in Medicine*, 15(1):53–76, 1999.

[78] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proceedings of the Conference on Deductive and Object-Oriented Databases(DOOD)*, Singapore, 1995.

[79] M. R. Quillian. Word concepts: A theory and simulation of some basic semantic capabilities. *Behavioral Science*, 12:410–430, 1967.

[80] D. Ragget. *HTML 3.2 Reference Specification*. W3C (World Wide Web Consortium), January 1997. At: http://www.w3.org/TR/REC-html32.

[81] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):126–149, May 1995.

[82] M. Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proceedings of 23rd International Conference on Very Large Data Bases*, 1997.

[83] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In R. T. Snodgrass and M. Winslett, editors, *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, pages 442–453, 1994.

[84] K. Stoffel, M. Taylor, and J. Hendler. Efficient management of very large ontologies. In *Proc. of Fourteenth American Association for Artificial Intelligence Conference (AAAI-97)*, Menlo Park, CA, 1997. AAAI/MIT Press.

[85] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures*. W3C, May 2001. At: http://www.w3.org/TR/2001/REC-xmlschema-1-20010502.

[86] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Rockville, MD, 1988.

[87] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, Rockville, MD, 1988.

[88] J. Vega, A. Gomez-Perez, A. Tello, and Helena Pinto. How to find suitable ontologies using an ontology-based WWW broker. In J. Mira and J. Sanchez-Andres, editors, *International Work-Conference on Artificial and Natural Neural Networks, IWANN'99, Proceeding, Vol. II*, pages 725–739, Berlin, 1999. Springer.

[89] W3C. *XHTML 1.0: The Extensible HyperText Markup Language*, January 2000. At: http://www.w3.org/TR/2000/REC-xhtml1-20000126.

[90] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3), 1992.

[91] G. Wiederhold. An algebra for ontology composition. In *Proceedings of the 1994 Monterey Workshop on Formal Methods*, pages 56–62. U.S. Naval Postgraduate School, 1994.

[92] G. Wiederhold. Interoperation, mediation, and ontologies. In *Workshop on Heterogeneous Cooperative Knowledge-Bases*, pages 33–48, Tokyo, Japan, 1994.