

Collaborative Ontology Construction for Information Integration

Adam Farquhar, Richard Fikes, Wanda Pratt, James Rice

Abstract

Information integration is enabled by having a precisely defined common terminology. We call this combination of terminology and definitions an *ontology*. We have developed a set of tools and services to support the process of achieving consensus on such a common shared ontologies by geographically distributed groups. These tools make use of the world-wide web to enable wide access and provide users with the ability to publish, browse, create, and edit ontologies stored on an *ontology server*. Users can quickly assemble a new ontology from a library of modules. We discuss how our system was constructed, how it exploits existing protocols and browsing tools, and our experience supporting hundreds of users. We describe applications using our tools to achieve consensus on ontologies and to integrate information.

The ontology server may be accessed through the URL

<http://www-ksl-svc.stanford.edu:5915/>

Keywords

Ontology, world-wide web interfaces, interoperation, knowledge reuse

1. Introduction	2
1.1. The Need for Ontologies	2
1.2. The Ontology Server Architecture	4
2. The Ontolingua Inclusion Model	5
2.1. Assembling an Ontology	5
2.2. A Semantic Model for Ontology Inclusion	8
2.3. Natural Input and Output	9
2.4. Summary	11
3. The Ontology Development Environment	12
3.1. Browsing Ontologies	12
3.2. Creating and Editing Ontologies	14
3.3. Maintaining Ontologies	15
3.4. Using Ontologies	15
3.5. Sharing Ontologies	15
3.6. Collaboratively Developing Ontologies	16
3.7. Ease of Use	16
4. Infrastructure	16
4.1. Layering State on HTTP	18
4.2. Adding Sessions, Users, and Groups	21
4.3. Encoding Requests and Results in URLs	21
4.4. Summary	24
5. Results	24
6. Conclusion	30
Bibliography	32

1. Introduction

1.1. The Need for Ontologies

In order for an agent to make statements and ask queries about a subject domain, it must use a conceptualization of that domain. A domain conceptualization names and describes the entities that may exist in that domain and the relationships among those entities. It therefore provides a vocabulary for representing and communicating knowledge about the domain.

Explicit specifications of domain conceptualizations, called ontologies, are essential for the development and use of intelligent systems as well as for the interoperation of heterogeneous systems. They provide the system developer with both the vocabulary for representing domain knowledge and a core of domain knowledge (i.e., the descriptions of the vocabulary terms) to be represented. Ontologies inform the system user of the vocabulary that is available for interacting with the system and about the domain and the meaning that the system ascribes to terms in that vocabulary. Ontologies are also crucial for enabling knowledge-level interoperation of agents, since meaningful interaction among agents can occur only when they share a common interpretation of the vocabulary used in their communications. Finally, ontologies are useful in many ways for human understanding and interaction. For example, they can serve as the embodiment of (and reference for) a consensus reached by a professional community (e.g., physicians) on the meaning of a technical vocabulary that is to be used in their interactions (e.g., exchange of patient records).

Ontology construction is difficult and time consuming. This large development cost is a major barrier to the building of large scale intelligent systems and to widespread knowledge-level interactions of computer-based agents. Since many conceptualizations are intended to be useful for a wide variety of tasks, an important means of removing this barrier is to encode ontologies in a reusable form so that large portions of an ontology for a given application can be assembled from existing ontologies in ontology repositories.

We have been working to develop and disseminate effective easy-to-use tools for creating, evaluating, accessing, using, and maintaining reusable ontologies (Fikes, Cutkosky, Gruber, & van Baalen, 1991). We have developed a set of tools and services to support not only the development of ontologies by individuals, but also the process of achieving consensus on common ontologies by distributed groups. These tools make use of the world-wide web to enable wide access and provide users with the ability to publish, browse, create, and edit ontologies stored on an *ontology server*. These tools and services provide many of the facilities that are crucial for promoting the use of ontologies and knowledge-level agent interaction including:

- A semi-formal representation language that supports the description of terms both informally in natural language and formally in a rigorous computer interpretable knowledge representation

language. We use an extended version of the Ontolingua language (Gruber, 1992) which provides a frame-like syntax in addition to full first order logic as specified in the Knowledge Interchange Format (KIF) (Genesereth, 1991). The language supports semi-formal definitions through the use of documentation strings and notes in addition to the formal specifications.

- Browsing and retrieval of ontologies from repositories. Browsing requires presentation of formal descriptions in an easily understood format. We make it easiest to express and browse knowledge that fits into an object-oriented frame view. We believe that the growing popularity of object systems (languages, databases, CORBA, etc.) substantially widens the group of people that are comfortable working in this paradigm. The key, however, is that the presentation of these objects is separated from their internal representation.
- Assembly, customization, and extension of ontologies from repositories. This requires the ability to identify and resolve name conflicts and to augment descriptions of terms from the assembled ontologies. We have extended our Ontolingua language so that users can quickly assemble a new ontology from a library of modules, as well as extend or restrict definitions from the library.
- Facilities for translating ontologies from repositories into typical application environments. We have developed translators from ontologies into a number of representations including CORBA's IDL (Mowbray & Zahavi, 1995), Prolog, CLIPS, LOOM (MacGregor, 1990), Epikit (Genesereth, 1990), KIF (Genesereth & Fikes, 1992).
- Facilities for programmatic access to ontologies so that remote applications have reliable access to up-to-date term definitions. We have defined a network protocol and application program interface (API) to enable remote applications to use an ontology server to learn about the vocabulary in an ontology and, for example, ask about the relations between terms.
- Support for distributed, collaborative development of consensus ontologies. We have developed a network accessible development environment (e.g., using Mosaic or NetScape) with a rich set of features to support concurrent ontology development such as fine-grained locking mechanisms and analysis of alternative definitions from multiple authors.

Ontology development and use technology will succeed when it becomes commonplace for people in a broad spectrum of communities to build and use ontologies routinely (e.g., as spread sheets are and e-mail is becoming). Another indicator of success will be the availability and widespread use of large-scale repositories of reusable ontologies in diverse disciplines (e.g., software development, database design and maintenance, network-based information retrieval, electronic commerce). These indicators of success should emerge when the technology has progressed sufficiently so that the benefits provided by using ontologies significantly outweigh the costs of developing them.

In Section 2 of this paper we discuss new features of the Ontolingua language that support assembly and reuse of existing ontologies from a repository. Section 3 addresses the design of our ontology editing tools. In Section 4, we discuss infrastructure that we have developed to provide collaborative ontology browsing and editing tools to a wide audience by exploiting existing protocols and browsing tools. Section 5 presents empirical evidence regarding the usability of the Ontology Server and our experience supporting hundreds of users.

1.2. The Ontology Server Architecture

Figure 1 shows a schematic view of the ontology editor and server. The leftmost box depicts the general-purpose ontology editor and server. The server provides access to a library of ontologies, allows new ontologies to be created, and existing ontologies to be modified. There are three modes of interaction with the Ontology Server indicated by the three boxes on the right side of Figure 1.

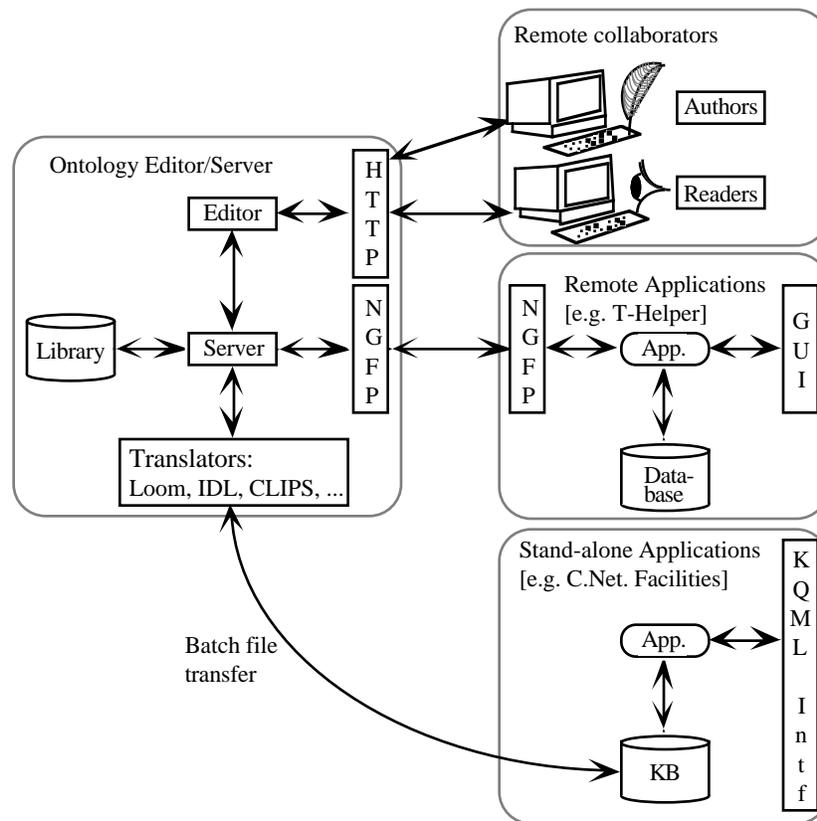


Figure 1: Architecture of the Ontology Server.

First, remote distributed groups can use their web browsers to browse, build, and maintain ontologies stored at the server. The server interacts with them through the now wide-spread hypertext transfer protocol (HTTP) and hypertext language (HTML) used on the world wide web. This makes the server accessible by a very wide audience. Any user familiar with common web browsing tools such as NetScape Navigator™

or NCSA's Mosaic can browse, build, and maintain ontologies stored at the server. The server allows multiple users to work simultaneously on an ontology in a shared *session*. The editor provides a number of features to support collaborative work (e.g., notifications, comparison, shared change logs).

Second, remote applications may query and modify ontologies stored at the server over the Internet. These programmatic interactions use a network API that extends the Generic Frame Protocol (Karp, Myers, & Gruber, 1995) with a network interface. The network interface supports queries (e.g., `is zidovudine an antiretroviral-drug`) as well as updates (e.g., `create a subclass of vector-quantity called 3d-vector-quantity`).

Third, a user can translate an ontology into a format used by a specific application (Gruber, 1993). The resulting translation can be used in a number of ways. For example, a CLIPS translation produces a set of class definitions and inference rules that can run directly in a CLIPS-based application. An Interface Definition Language (IDL) translation produces an IDL header file that a CORBA compliant program can use to interact with an Object Request Broker (ORB). A KIF translation produces a file of logical sentences that can be used by a logic-based facilitator, such as the one fielded by CommerceNet to draw inferences in response to client queries and to route these queries correctly.

2. The Ontolingua Inclusion Model

2.1. Assembling an Ontology

We want ontologies to be practical and useful artifacts. This means that the effort required to construct new ontologies must be minimized and the overall effort required to construct an ontology must be amortized over multiple uses and users. We are taking several steps towards attaining that goal. One is to provide a rich editing environment that provides automated analysis capabilities. A second is to make this editing environment readily accessible to a wide community. A third is to provide explicit support for collaborative ontology construction. A key fourth step is to enable ontology writers to reuse existing ontologies in a flexible and powerful way. In this section, we show how the Ontology editor allows users to reuse existing ontologies from a modular structured library by inclusion, polymorphic refinement, and restriction.

Formally, ontologies in our system are specifications of axiomatic logical theories. An ontology specification consists of a vocabulary of non-logical symbols and a set of KIF axioms that constrain the acceptable referents of those symbols. An Ontolingua user specifies these axioms in the form of definitions of classes, relations, functions, and constants using frame language constructs where appropriate. *Non-logical symbols* are the atoms that name relations, functions, and constants. In this section, we describe our approach to providing ontology reuse in terms of operations on sets of axioms and the non-logical symbols that occur in them.

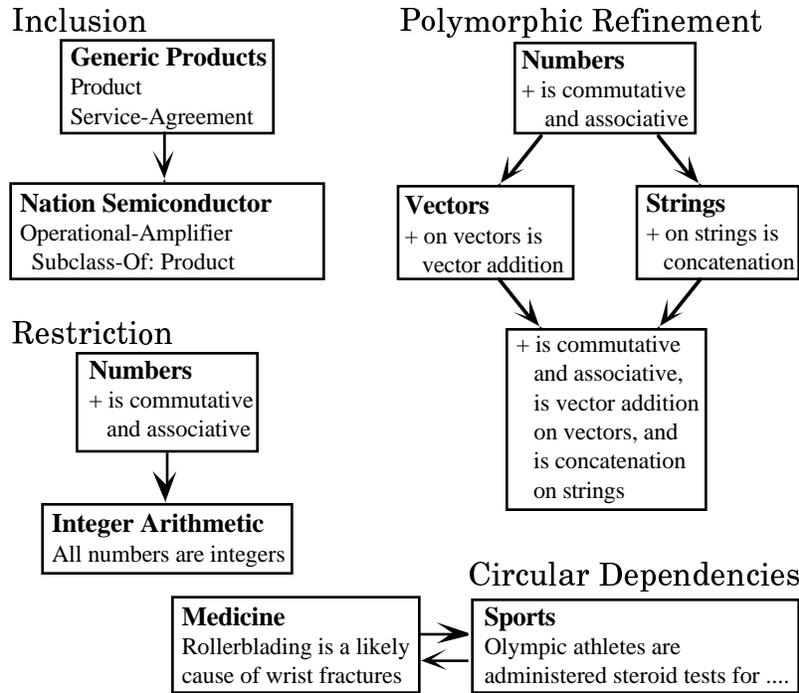


Figure 2: There are many relationships between ontologies.

Figure 2 shows several motivating examples that are drawn from our ontology building experience. Example 1 shows the simplest relation between ontologies: *inclusion*. The author of a National Semiconductor product ontology needs to represent basic information about products, prices, services, and so on. She wants to include the entire contents of the generic product ontology from the ontology library without modification.¹ In Example 2, the author wishes to extend the addition operator + in two distinct ways. The library contains axioms about the addition operator in the KIF-numbers ontology (e.g., it is associative, commutative, has 0 as an identity element, etc.). She wishes to extend the addition operator to apply to vectors in one ontology and to apply to strings in another ontology — we refer to this operation as *polymorphic refinement*. In Example 3, we see that the inclusion relations between ontologies may be circular. We consider two ontologies: one for medicine and another for sports. The medical ontology will need to refer to a variety of terms from the sports ontology (e.g., roller-blading is a leading cause of wrist fractures in teens) and the sports ontology must also refer to medical terms (e.g., weight-lifters may use anabolic steroids to increase muscle growth). We must handle this sort of relationship carefully because the ontology designers do not want either ontology to be polluted by the non-logical symbols from the other. In Example 4, we see that specialized ontologies may make simplifying assumptions that *restrict* the included axioms. For example, in the integer-arithmetic ontology, all numbers are restricted to be integers.

¹Notice that by “inclusion” here, we do not mean “cut and paste the contents of the product ontology into the HP ontology file”. This interpretation would result in unfortunate version dependencies.

Many knowledge representation systems have addressed these issues in one way or another. Before turning to our solution, we will discuss some of the approaches that others have used, illustrate some of their shortcomings, and use them to motivate our novel design choices.

The easiest and simplest approach is to provide no explicit support for modularizing represented knowledge — let the author beware. For instance, the THEO system (Mitchell et al., 1989) uses a single knowledge base and a single set of axioms. In some sense, this enables Examples 1-3 to be represented, but it has two key drawbacks: First, it is impossible to restrict definitions (Example 4). Second, by eliminating modularity, it makes understanding and evaluating ontologies a nightmare. Authors using systems like this often resort to lexical conventions to discriminate between symbols (e.g., `+@kif-numbers`, `+@vectors`, `+@strings`). Without automated support, such conventions are difficult to enforce. Furthermore, enforcing them may not even be desirable. In Example 2, the axioms in the `vectors` ontology are about the same `+` operator as the axioms in `kif-numbers`.

A fairly common extension is to allow a directed acyclic graph (DAG) of inclusion relations between “theories” such as provided by Genesereth’s Epikit. That mechanism supports modularity, restrictions, and incompatible augmentations. It has two drawbacks: First, no cycles are allowed among theories. As we have seen, it is both natural and desirable to have cyclic relationships between terms in ontologies.² Second, in its simple form, this mechanism results in unnecessary name conflicts. For instance, an ontology for scientific course work might include ontologies for chemistry and academics, both of which define `tests`, but in different ways. There must be a way of discriminating between `tests` in chemistry and `tests` in academics.

The LOOM system provides a DAG of inclusion relationships, but extends the simple approach by allowing references to non-logical symbols in ontologies that have not been included. Referencing a symbol in an unincluded ontology, however, does not include all of the axioms from that ontology, but only minimal type information. This conflates the declarative semantics, as defined by the axioms, with pragmatic information about which axioms to apply during problem solving.

There are two distinct aspects to our solution to these issues: (1) the inclusion model which defines how axioms and non-logical symbols are included in new ontologies, and (2) the interface model which defines the relationship between character strings input by (or output to) the user and non-logical symbols in the ontologies.

²Indeed, we initially wanted to avoid the additional complexity introduced by allowing circular references, but our users demanded it. For any particular example in which circular references occur, it is always possible to create a new ontology, (e.g., sports-medicine) that contains the subset of the ontologies in the cycle. This is not a practical solution, however, because it may require the entire structure of the ontology library to be changed to add a single axiom. This would make it impossible to have a general-purpose library of ontological fragments that can be reused.

2.2. A Semantic Model for Ontology Inclusion

In order to facilitate the reuse of existing ontologies, the Ontology Server provides a facility for *including* one ontology in another as follows. Each ontology is considered to be specified by a vocabulary of non-logical symbols and a set of axioms. Formally, including an ontology A in an ontology B requires specifying a translation of the vocabulary of A into the vocabulary of B, applying that translation to the *axioms of A*, and adding the translated axioms to the *axioms in the specification of B*. We say that the axioms in the resulting set are "the axioms of ontology B" so that if ontology B is later included in some other ontology C, the resulting ontology C will include translated versions of both the axioms in the specification of B and the axioms of A. Thus, when we say "the axioms of ontology O", we mean the union of the "axioms in the specification of O" and the axioms of any ontology included by O. This notion of inclusion defines a directed graph of inclusion relationships that can contain cycles. We allow ontology inclusion to be transitive and say that ontology A is included in ontology B if there is a path in the ontology inclusion graph from A to B.

The Ontology Server eliminates symbol conflicts among ontologies in its internal representation by making the symbol vocabulary of every ontology disjoint from the symbol vocabulary of all other ontologies. That is, in the internal representation, the symbol S in the vocabulary of ontology A is a different symbol from the symbol S in ontology B.

Given that symbol vocabularies are disjoint, the Ontology Server can assume in its internal representation that the translation used in all inclusion relationships is the identity translation. Therefore, in the internal representation, including an ontology A in an ontology B simply means adding the axioms of A to the axioms of B.

Note that in this simple model of ontology inclusion, cyclic inclusion graphs are not a problem since the only effect of ontology inclusion is the unioning of sets of axioms.

If an ontology A contains an axiom that references a symbol in the vocabulary of an ontology B, then B is considered to be included in A. The Ontology Server allows users to state explicit inclusion relationships between ontologies and implicitly creates inclusion relationships based on symbol references in axioms.

2.3. Natural Input and Output

"... no more complex than the Common Lisp package system"

— James Rice

The semantic model introduced above provides a powerful, simple, and unambiguous way for ontologies to be assembled and reused. However, in order to eliminate ambiguity, it requires symbols to be given clumsy extended unique names that may be unknown to the user. Moreover, it does not allow a user to perform important operations such as renaming symbols from included ontologies or selectively controlling which

symbols are to be imported from included ontologies or exported to other ontologies. The Ontology Server solves those problems with additional capabilities that are a part of its facilities for converting symbol references in input/output text to and from the internal symbol representation. We describe those capabilities in this section.

Ontolingua requires that any non-logical symbol referred to in an input or output stream be *defined* in some ontology and be assigned a *name*. The ontology in which a symbol is defined is called that symbol's *home ontology*. Similarly, each ontology has a name that uniquely distinguishes it from any other ontology.

The Ontology Server interprets a symbol reference in an input stream or produces a symbol reference in an output stream from the perspective of a given ontology. For example, if the symbol *S* is defined in ontology *A* and also defined in ontology *B*, then from the perspective of ontology *A*, the input text "*S*" is interpreted as "the symbol named *S* defined in ontology *A*", whereas from the perspective of ontology *B*, the input text "*S*" is interpreted as "the symbol named *S* defined in ontology *B*".

The perspective from which any given symbol reference is to be interpreted can be explicitly specified by attaching a suffix to the symbol name consisting of the character "@" following by the name of an ontology. So, for example, the symbol named *S* interpreted from the perspective of ontology *A* can be explicitly referred to as "*S@A*". A symbol reference that includes the @«ontology name» suffix is said to be a *fully qualified reference*. Fully qualified references enable symbols defined in any ontology to be referenced in any other ontology.

The Ontology Server input/output system provides a symbol renaming facility that allows a user to assign a name to a symbol which is *local* to the perspective of a given ontology. A renaming is specified by a rule that includes an ontology name, a symbol reference, and a name that is to be used as the local name of the given symbol from the perspective of the given ontology. Given such a renaming rule, the system will recognize the local name as a reference to the given symbol when processing input in the given perspective, and will use the local name to refer to the given symbol when producing output from the given perspective. So, for example, a renaming rule might specify that in ontology *A*, the local name for *auto@vehicles* is to be *car*. This facility enables an ontology developer to refer to symbols from other ontologies using names that are appropriate to a given ontology and to specify how naming conflicts among symbols from multiple ontologies are to be resolved.

For convenience of input and parsimony of output, the @«ontology name» suffix can be omitted from symbol references when the symbol name itself is unambiguous from the intended perspective. In particular, a given symbol can be referred to from the perspective of an ontology *A* simply as *S* if and only if the given symbol:

- Is defined in ontology *A* with name *S*;
- Has been renamed to *S* in ontology *A*; or

- Has been *imported* into ontology A.

A name that can be used to refer to a defined symbol from the perspective of a given ontology is said to be *recognized* in that ontology. Thus, the convention given above for omitting the @«ontology name» suffix from symbol references implies that a name S is recognized in an ontology A if and only if S is the name of a symbol defined in A, or S is the name of a symbol that has been imported into A and has not been renamed in A, or S is the local name for a symbol in A.

We now describe the mechanisms for importing symbols into an ontology.

Each defined symbol is designated as being *public* or as being *private* to the ontology in which it is defined. The system considers symbols to be public by default so that users can ignore the public/private distinction until they encounter or want to define private symbols.³

The Ontology Server associates with each ontology a set of ontologies whose public symbols are imported into the ontology. User commands are available for editing that set of ontologies. However, in order to simplify the use of this symbol importing mechanism, by default the Ontology Server automatically adds to this set any ontology that is explicitly included. Thus, users can ignore the distinction between symbol importing and explicit inclusion until they want to override that default.

In order to provide control over the importing of individual symbols, the Ontology Server associates with each ontology a set of *shadowed* symbols that are blocked from being imported into the ontology. That set of shadowed symbols overrides symbols in the set of imported ontologies in that a symbol is imported into an ontology A only if the symbol is a public symbol defined in an ontology that A imports and is not shadowed in A, or the symbol is a public symbol in an ontology that does not import, but is imported by means of an identity renaming (e.g., S@A goes to S@B)

The Ontology Server uses the shadowing mechanism to prevent ambiguities from occurring in the text form of symbol references by automatically blocking the importation into an ontology of any symbols which would have the same text form in that ontology's perspective. Thus, if there is a symbol that can be referred to by S from the perspective of ontology A, a different symbol that can be referred to by S from the perspective of ontology B, and the public symbols from both ontologies A and B are imported into ontology C, then the Ontology Server automatically adds S@A and S@B to C's list of shadowed symbols in order to prevent "S" from being an ambiguous symbol reference from the perspective of ontology C. This automatic shadowing occurs irrespective of the order in which the definitions and inclusion relationships are specified. So, for example, if the inclusion relationship between ontologies A and C already exists when a symbol that can be referred to by S@A is defined, then S will be added to the set of symbols shadowed in ontology C at the time S@A is defined.

³Users can change the default on a per-ontology basis.

2.4. Summary

To summarize, we note how Ontolingua supports ontology inclusion, circular dependencies, and polymorphic refinement by reconsidering the examples discussed earlier from Figure 2.

Using Ontolingua, the ontology inclusion relationship in Example 1 would be explicitly established by the developer of the National Semiconductor Products ontology either as part of the definition of that ontology or as an editing operation after the ontology has been defined. Given that inclusion relationship, public symbols from the Generic Products ontology, such as `Service-Agreement`, whose names do not conflict with other recognized names in the perspective of the National Semiconductor ontology would, by default, be imported into the National Semiconductor Products ontology and therefore could be referred to from the perspective of that ontology by their names (e.g., `Service-Agreement`) without the `@GenericProducts` suffix.

The circular dependencies in the `Medicine` and `Sports` ontologies of Example 2 would be established and presented by using fully qualified names to refer to symbols from the perspective of the other ontology. For example, in the `Medicine` ontology, roller-blading would be referred to as `roller-blading@sports`, and in the `Sports` ontology, steroid tests would be referred to as `steroid-tests@medicine`. The reference to roller-blading in the `Medicine` ontology will cause the axioms of the home ontology of the symbol `roller-blading@sports` to be implicitly included in the `Medicine` ontology, but would not cause the public symbols from the `Sports` ontology to be imported into the `Medicine` ontology.

The polymorphic refinement of `+` in Example 3 is a case in which some of the subtle properties of implicit ontology inclusion become apparent. If ontology `X` does not explicitly include the `vectors`, `strings`, or `numbers` ontologies, then references to `+#@vectors` and `+#@strings` in `X` will cause `numbers` to be implicitly included in `X`, but will not cause `Vectors` or `Strings` to be included since both `+#@vectors` and `+#@strings` refer to a symbol whose home ontology is `Numbers`. If the vector addition axioms of ontology `Vectors` and/or the string addition axioms of ontology `Strings` are to be included in ontology `X`, then the user must state the inclusion relationship explicitly.

3. The Ontology Development Environment

Our goal was to create a general environment to facilitate the development and sharing of ontologies. Such an environment must assist the user in the basic development tasks of browsing, creating, maintaining, sharing, and using ontologies. We also realized that many of our users desire to develop ontologies through a consensus process; therefore, we also needed to provide tools to help people collaborate during the development of their ontologies. In this section, we will discuss the features implemented in our web-

based ontology environment which provide assistance with basic development tasks, facilitate collaboration, and improve ease-of-use.

3.1. Browsing Ontologies

An essential component of our browsing environment is being able to quickly jump from one term in the ontology to another using hyperlinks. Selecting the name of a term takes the user to a page displaying the definition of that term. The definition consists of both informal documentation and formal statements about that term. Rather than displaying this information in a purely logic-based form, we display this information in an object-oriented or frame-based form (see Figure 3). In a frame, slots are displayed along the left side of the screen and values corresponding to the slots are displayed following the name of the slot. Information that can't be displayed as slot and facet values appears later in the page as axioms. The axioms are split into three categories: equivalence axioms, implication axioms, and other related axioms.

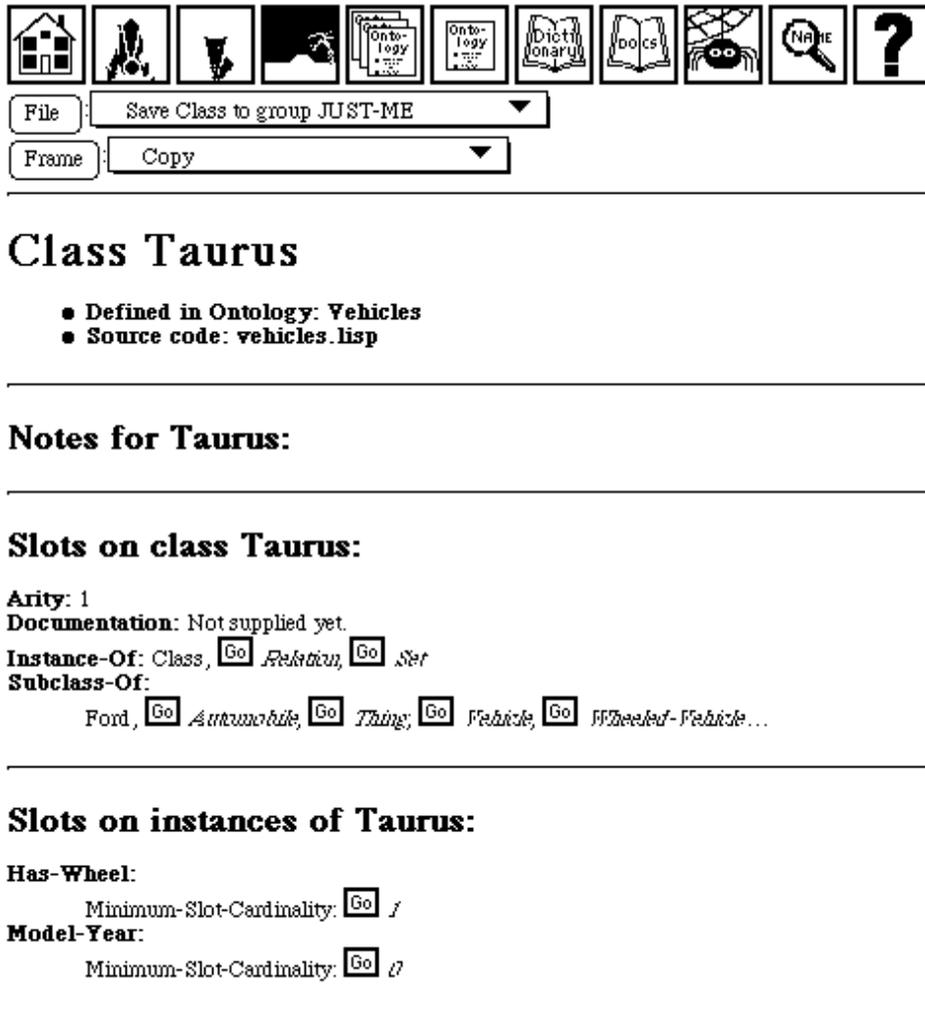


Figure 3: A screen image from the ontology editor showing the class definition for Taurus cars in the vehicles ontology.

When users examine an entire ontology, they often want to see information about a term that is inferable from the definition of other terms. To support this feature, the ontology server performs a limited set of inferences that provide the inferred information that is typically provided by frame-based representation systems. In particular, the ontology server supports class-subclass and class-instance inheritance, slot inverses (e.g. (subclass-of Taurus Ford) implies (superclass-of Ford Taurus)), and the semantics of slots that are predefined as part of the frame language (e.g. (subclass-of Taurus Ford) and (subclass-of Ford Automobile) implies (subclass-of Taurus Automobile)). To distinguish between inferred and directly asserted information, the direct information appears in a normal font, and the inferred information appears in italics. The ontology server also supports a way for the user to see where

the inferred information was directly asserted by providing a button labeled “GO” in front of each piece of inferred information.

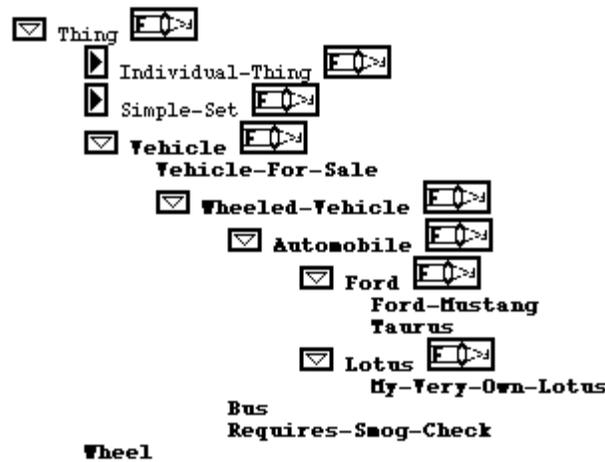


Figure 4: The class browser provides an overview of the class hierarchy.

When the user is not interested in seeing information about each term in great detail, a view can be obtained of the general hierarchy of classes. To support this functionality, we have implemented a feature where the user can view all of the classes in an ontology and how they are hierarchically related to other classes in the ontology library (see Figure 4). Classes appear indented under their superclasses. Each class that has subclasses is prefaced by an arrow which when pointing downward indicates that all of the subclasses of the class are to be displayed, but when pointing horizontally indicates that only the class is to be displayed. In addition, when the user wants to focus on a particular class, they can choose the “focused browse” option and view the selected class with only its superclasses and subclasses displayed.

3.2. Creating and Editing Ontologies

We designed the ontology editing environment to have a similar appearance to the ontology browsing environment so that the user would not need to learn two different interfaces. The difference between the editing environment and the browsing environment is that two new types of icons appear in the editing environment: edit pens and insertion icons. Edit pens appear in front of information that can be modified by the user. When the user wishes to change that information, she selects the appropriate edit pen, and a form for modifying the current information appears. Insertion icons appear wherever a user is allowed to add information such as a new value or facet for a slot. When the user selects one of the insertion icons, a form for entering new information appears. The contents of these forms depends on the type of thing which is being added or edited.

Commands which don’t fit into this paradigm appear as menu options at the top of the screen. For example, commands for creating new terms appear on these menus. The user can select which type of term

to create, and a form tailored to that type of term will prompt the user to fill in information about the new term.

3.3. Maintaining Ontologies

The ontology server provides several features to assist with the maintenance of ontologies. For examples, users can compare an ontology to previous versions of that ontology. This can be very useful if the user wants to monitor changes in the ontology or even undo some of the changes. Ontologies can also be analyzed for inconsistencies using the “Analyze Ontology” command. All slots, slot values, facets, and facet values are checked to make sure that they conform to the constraints that apply (e.g., domain, range, slot value type, and cardinality constraints).

Informal documentation is an important part of making ontologies maintainable. The ontology server supports special key words within the informal documentation known as notes. A user can assign a keyword such as example, verified-by, see-also, modification-by, and formerly-named to provide more structure to the informal documentation.

3.4. Using Ontologies

Although the ontology server currently does not provide much inferential power, it does provide some support for using ontologies. One way to use ontologies developed with the ontology server is to translate the ontology into the representation language of another system such as CLIPS, LOOM, or Prolog. Currently, the ontology server can translate into ten different representations. Users can then transfer the translation via e-mail. Users may also e-mail their ontologies as standalone hyperwebs, source code, or formatted text.

3.5. Sharing Ontologies

The ontology server provides a number of features to promote the sharing of ontologies. The primary mechanism for supporting ontology reuse is through the library of ontologies which acts as a central repository for reusable ontologies. When someone has an ontology that they believe is ready for others to use, they can choose to publish their ontology in the ontology server. After the ontology has been approved, it becomes available to anyone with access to the ontology server through the library.

The ontology server also provides several tools for searching for terms within ontologies in the library. A user may choose the general search facility which allows them to use wild-cards in searching the entire library for terms whose name matches the specified pattern. Context-sensitive searching is also available whenever the user needs to fill in the name of a term such as when adding a value to a slot. In context-sensitive searching, constraints on which terms are appropriate are used to limit the search.

3.6. Collaboratively Developing Ontologies

One of the features which clearly distinguishes the ontology server from other ontology development environments is its support for collaborative ontology construction. The ontology server uses the notion of users and groups that is typical in most multi-user file systems. As with file systems, read and write access to ontologies is controlled by the ontology owner giving access to specific groups. This mechanism supports both access protection as well as collaboration across groups of people who are defined within the ontology development environment.

Ontology server users supply their user name and password at the beginning of each session. A user may also specify that she wants her session to be accessible to other members of a specified group, she wants to connect to an already active group session, or she wants to start her own private session. When a session is shared among a group, modifications that any member makes is visible to the other members of the shared session both as changes to the ontology and through textual notifications of the changes that are sent to every other member of the session.

3.7. Ease of Use

In designing the interface to the ontology server, we wanted to make a tool which would be simple for a novice to understand and use yet be powerful enough to support experienced users. To accommodate such differing levels of users, we added a large variety of user preferences for controlling the behavior of the user interface.

To make the interface simple, the ontology server provides four basic types of pages: the table of contents for the ontology library, ontology summary pages, frame pages (for classes, relations or instances), and the class browser. The ontology server provides a wide variety of other features to make the environment easy to use. The hyperlinked environment makes it easy to provide tutorials, online documentation, and context-sensitive help that can be selected at any time. In addition, the server allows the user to undo or redo any number of modifications they made to the ontology since they last saved it.

4. Infrastructure

In this section, we discuss the infrastructure necessary to provide a tool that will support distributed collaborative work on ontologies. One of our primary goals has been to make access as easy and wide-spread as possible. This means that we must support access from a wide variety of platforms (e.g., PC, Mac, and UNIX) in a wide variety of settings (e.g., academic, research, and industrial). This imposes strong constraints on any actual implemented tools. To achieve this goal in a cost effective way, we have chosen to leverage the existing (and developing) World Wide Web (WWW) infrastructure.

The WWW provides distributed access to multi-media hypertext documents. Its infrastructure consists of four components: clients that run on every user's machine, servers that run on information provider machines, the hypertext transfer protocol (HTTP), and the hypertext markup language (HTML). Clients include NetScape Navigator, NCSA's Mosaic, the text-based LYNX, and others. They are robust, run on all platforms, and are already installed and in daily use by millions of potential users. HTTP specifies how a client requests a document and how a server returns the document to the client. HTML enables documents to be "marked up" with formatting information as found in most word processors (bold face or italics, section headings, etc.) and also hypertext links that are associated with specific portions of text. The links indicate a new document that is to be retrieved when a user selects the associated text. A key component of HTTP/HTML is the uniform resource locator (URL) that specifies the location of a hypertext document. The URL includes the access protocol, hostname, port, and a string that specifies a "location" on the particular host.

We have leveraged the WWW infrastructure by: (1) letting users view an ontology as a hypertext document with their WWW clients, and (2) constructing a special server that provides them with editing capabilities. This achieves our goal because the WWW clients are wide-spread, run on most platforms, and shelter us from different platforms, interfaces, and operating systems. Furthermore, because many of our users will already be familiar with their WWW clients, they can easily integrate our tools into their daily practice.

There is, however, a catch. HTTP was originally designed to request and describe "static" hypertext documents. The WWW community has extended HTTP/HTML to include more interactive features and better support for dynamic document generation so that it provides better support for sophisticated applications such as the Ontology Server. But several problems with HTML/HTTP must still be overcome. This section will focus on overcoming some of the inherent restrictions in HTTP.

HTTP is a *stateless transaction-based* protocol. By *stateless*, we mean that the server need not store any information about a client or its requests. This means that every request is completely self contained; it includes all the information needed by the server to answer the request. By *transaction-based*, we mean that the fundamental element of interaction is a simple transaction in which the client opens a network connection to a server, sends a single request over the connection, receives a response, and the connection is closed. This may be contrasted with session-based protocols in which connections persist over many transactions. The decision to make HTTP a stateless transaction-based protocol was a good one. It means that servers can be more efficient and robust — they do not have to retain information about clients or connections to them. Servers that process hundreds of thousands of requests from tens of thousands of clients per day could hardly operate any other way.

The requirements on a protocol that will support distributed collaborative editing, however, are very different. In particular, a distributed collaborative editing environment must provide a *stateful connection-*

based protocol. One of our challenges has been to define and implement a stateful connection-based protocol that is layered on top of HTTP in a way that is transparent to existing browsers.

4.1. Layering State on HTTP

There are three broad approaches to representing state in a stateless transaction-based protocol:

- (1) Encode the state in each request and response. In this approach, the client is responsible for retaining the state information; the server need only encode the current state in the response and decode it when processing a request.
- (2) Include a time-stamped token in each transaction indicating the state when the transaction occurred. The server retains the state together with the token. If the server retains old states (e.g., by storing a table of states or retaining replayable transaction records), then this method allows for a kind of time-travel. By re-executing an old request, you can return to the state that originally resulted from its execution. Furthermore, requests are idempotent — executing the same request multiple times has no additional effect.
- (3) Include enough information in the request to identify the portion of the state that is affected uniquely. Requests are always evaluated against the current state, which is stored by the server. This approach does not allow time-travel. Executing the same request may have cumulative effects.

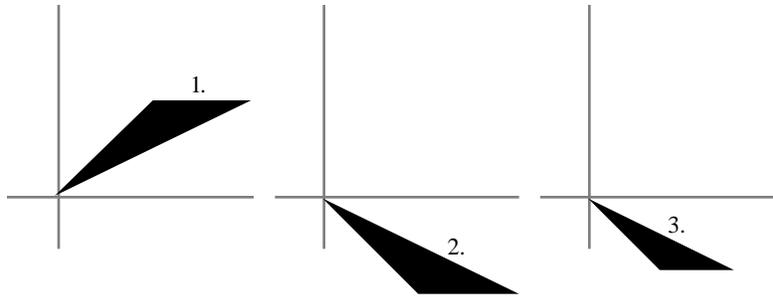


Figure 5: A triangle after being flipped and scaled as a result of client requests.

(1) Client State	(2) Server Tagged State	(3) Server State
obj=[(0,0)(1,1)(2,1)], command=flip-vertical	obj=1, command=flip- vertical	obj=1, command=flip- vertical
obj=[(0,0)(1,-1)(2,-1)]	obj=2	obj=1
obj=[(0,0)(1,-1)(2,-1)], command=scale75	obj=2, command=scale75	obj=1, command=scale75
obj=[(0,0)(.75,-.75)(1.5,-.75)]	obj=3	obj=1

Table 1: Requests and responses illustrating the differences between three possible approaches to representing state in a stateless protocol.

Table 1 and Figure 5 illustrate the differences among these three approaches. They show how three different servers might deal with the problem of flipping and scaling a triangle. In approach 1, the `obj` field contains a list of the points that define the polygon. Sending the request that defines the triangle together with a `flip-vertical` command returns a new representation of the flipped object. Sending the next request includes the object representation along with the scale command and the scale factor as an additional argument. The server returns a specification of the scaled flipped polygon. A number of WWW servers employ this approach. When the state is large or does not admit a simple printed representation, measures must be taken to find a more compact and printable representation (e.g., the polygon being transformed might have thousands of points). Two common methods are (1) to apply a compression algorithm to the representation, and (2) to use a sequence of transformations applied to a reference object (e.g., the Xerox PARC Map server). For many applications, however, this approach is not feasible. Imagine an editor constructed using this method — it would transmit an entire document twice with each editing change.

In approach 2, the object is not explicitly represented in the request or response. Instead, each message contains a token indicating to the server which version of the world the operation is applied to. `obj=1` refers to the original triangle, `obj=2` the flipped triangle, and `obj=3` the scaled, flipped triangle. One nice property of this representation is that it allows time travel. For example, the user could execute a `scale50` command to `obj=2` resulting in `obj=4` in which the triangle is flipped and then scaled 50%. If the user

interacts with a web-browser, reloading a page will always produce the same results. There are two problems with this: First, it may be too expensive for the server to retain or represent all the states. If old states were not recorded, reloading a page would result in an error indicating that the referenced state is no longer available. Second, many users may find time travel somewhat surprising. For example, suppose that our user with a web-browser revisited the page with the original triangle and reloaded it, the image would be unchanged. Many users might interpret this as indicating that their changes had been lost, which could be quite disconcerting.

In approach 3, requests are always evaluated against the current state. Instead of a state token, a token is present for each argument that is needed to execute the command. Note that the responses from the server may differ after each command is executed. If the user were using a web-browser, backing up to an old image of the triangle and reloading it after the transformations would result in seeing the transformed image. This approach requires less work on the part of the server, because only the current state is retained. It may also be more intuitive for many users.

The Ontology Server uses this last approach. Every request is evaluated against the current state. Each request encodes both the command name and its arguments.

Server-side persistence of user state is necessary to implement this scheme (there has to be persistent state). We chose to use a persistent server. The HTTP protocol is still simple enough to make this practical. Alternatives are to use a persistent database store of some sort, or to provide a transient gateway from a standard HTTP server that would pop up a connection to the persistent non-standard server.

Our strategy is different from that used by most HTTP servers, such as NCSA's HTTPD. A typical HTTP servers listen for network connections, and upon receipt of a connection, fires up an ephemeral process to handle the request. These processes have their own address spaces and have no state available to them other than the arguments passed to them in the HTTP request. Our server, in contrast, has a single, persistent process listening for connections. When a request comes in, it fires up an ephemeral *thread* within itself to handle the request. The thread has access not only to the arguments in the request, but also to all of the state in the shared address space of the server.

The *session* is the primary state representing object. Individual pages may also have state. User state (preferences) is orthogonal to sessions. Separating out sessions from users allows a many/many mapping of sessions to users. A consequence is read/write locking. We do not have deadlock because there is no checkout (problematic in a transaction based protocol) of pages. Writes occur in sequence. User notifications, comparison, undo, and redo help to avoid problems. Our application does not tend to result in multiple simultaneous attempted edits of the same object.

4.2. Adding Sessions, Users, and Groups

State is not enough to support collaborative editing. We also need representations for users and sessions so that each user can customize her environment and groups of users can work collaboratively on a shared object, (e.g., an ontology). A session provides more than a grouping of transactions. Multiple users may share state within a session. The server must maintain session-specific state information and keep the contents of each session distinct and inviolable by other sessions.

There are several possible approaches to handling the session-specific state:

- Spawn a process with its own address space for each session.
- Use a persistent store to hold all of the state information for each session, updating and retrieving from the appropriate store to handle each request. This way no state is cached in the server.
- Use a session data structure to store the state information and avoid inappropriate pointers from one session to another.

The application domain determines which choice is most appropriate. The Ontology Server uses the last method for three reasons. First, there is a large library of read-only ontologies that are included (perhaps transitively) in most user ontologies. Using a single address space makes it much easier for these objects to be shared across sessions. Second, the complete environment including the editor, translators, analysis tools, and so on comprise a substantial system that would be needlessly duplicated in spawned processes. Third, using a single server process means that we can easily patch, modify, or extend it while it is running. Indeed, the vast majority of such changes have been introduced without interrupting service to our users.

When a session is created, the server builds a session object in which the session-specific state is stored and generates a unique session identifier. Once the session has been created, all requests that change or examine the session-specific state must include the session identifier. sers of a session.

The last piece in the puzzle is the user. The server supports individual users with a database of user preferences. When a user performs a password protected login, she may either create a new session or enter an existing session which was created with group access.

4.3. Encoding Requests and Results in URLs

In the previous section, we describe three general methods for layering state on a stateless protocol such as HTTP. In this section, we describe the precise mechanism that our server uses. This mechanism is widely applicable to systems that want to provide stateful interaction over the WWW.

Every URL represents a complete procedure call — a procedure to be executed together with all of the arguments that the procedure requires. These procedures emit HTML pages, which may include embedded URLs that result in additional procedure invocations when selected. Instead of encoding the procedure and

all of its arguments in the URL, we use a unique token. This token is then used as an index into a table that is stored within the server. Because the server also has session and user specific state, the URL will have three parts: the session identifier, the user identifier, and the page command identifier.

Consider the problem of providing web access to a relational database of bibliographic information. The first page that a user sees displays a form that includes an entry field for a query and a submit button to execute it. The titles of the first few matches are returned on the second page; selecting a title causes the full citation to be displayed. At the bottom of the second page is a button to get more titles.

Let's see how this can be supported by a UID table. The table must contain all of the information necessary to determine the contents of each page uniquely. One design for this table is shown in Table 2 .

UID	Command	DB	Start	Query
UID0	get-query	—	—	—
UID1	execute-query	db1	0	select title from bib where author = "Fikes"
UID2	execute-query	db1	2	select title from bib where author = "Fikes"
UID3	get-citation	db1	—	select citation from bib where no = 25
UID4	get-citation	db1	—	select citation from bib where no = 37

Table 2 : The UID table associates unique identifiers, which are used in URLs, with a command and its arguments.

The URLs are formed by combining the UID with the session and user identifiers. For example, the URL for the initial database query might be `/sid27/uid1&user=farquhar`. Note that if another user added citations for "Fikes" and the first user were to reload the page with the citations listed, it would be updated with the new citations. Furthermore, the server does not need to store all of the information returned by the query — only the commands that would access the additional information that the user might need. If the `execute-query` command recognized a user preference that determined whether the titles or the full citations would be returned, then changing the preference and reloading the page would result in the correct behavior of reporting the full citations.

Our current server implements this approach. There is a global table of session objects; each session object contains a table of users who have joined the session, the session-specific state, and two tables to implement the UIDs. The first UID table maps from UIDs to *item* data structures that contain the command and argument information (see Table 2), as well as a pointer back to the UID. The second table maps from the command and argument information to an item, and consequently to the UID. Each of these

are implemented as hash tables to provide rapid access, although the first table could also be implemented as an extensible array indexed by numeric UIDs.

One other approach would be to allocate a different UID for every occurrence of a link on a page irrespective of whether it had been encountered before. This obviates the need for a reverse mapping table, but in practice would vastly increase the size of the UID table and would have the undesirable property that the user could not perceive the right result in her browser. In the case with the reverse mapping table, every link denoting a given command would change color after being selected by a user. In the latter case, this would not happen and would leave the user confused, thinking “I could have sworn I had been there, but it hasn’t turned red”.

We can see how these tables and data structures are used to process some of the queries in the above example. When the user submits the query form, her web client transmits the URL:

```
/sid27/uid0
&user=farquhar&db=db1&command=executequery
&query=select+title+from+bib+where+author=fikes
```

The server extracts the session, unique, and user identifiers. The session identifier is used to fetch the session object and establish any global (or dynamic) variable bindings from it. The same is done with the user identifier. Next, the session’s UID-to-item table is used to fetch the item corresponding to UID0. The corresponding `get-query` command is executed with the `db` and `query` arguments that are extracted from the URL. The `get-query` command uses the `db` and `query` arguments to construct a key for the second table to see if there is already a UID for this query. There is not, so it creates UID1, executes the query and creates UID2, UID3, and UID4 during the process of constructing the HTML page for the results. This result page will contain the URL `/sid27/uid3&user=farquhar` which, when selected, will execute the command that will get citation 25. We can think of this URL as being anchored to the citation, though an arbitrary amount of computation is performed in processing the `get-citation` command, and the citation might not exist as an HTML document. The page associated with this URL may well be dynamically constructed by the server. If the user selects this URL, the server will establish the session and user information as before. Next, it simply fetches the item for `uid3` from the session’s UID-to-item table and executes the stored `get-citation` command.

A robust server must, of course, validate the session, user, and unique identifiers, and report any command execution errors in a graceful fashion. Our server is implemented in Common Lisp. This meant that many of the underlying mechanisms required for implementing this approach were already in place including hash tables that work on arbitrary structures, dynamic bindings, and a powerful condition handling system.

The cost of this approach appears to be very reasonable for our applications. In practice, the UID table is extremely sparse. Its size is bounded by the number of URLs that are actually presented to the user. On the

one hand, in the Ontology Server every page is heavily hyperlinked (almost every word on a page is hyperlinked). This suggests that many UIDs will be created. On the other hand, many of the hyperlinks appear multiple times and commonly reappear on other pages. As a result, the number of entries in the UID tables is small — on the order of hundreds or a few thousands. Furthermore, the granularity of pages and commands is sufficiently large that users look at a fairly small number of pages per day (c.f. a text editor interface that updates a document at the paragraph level, rather than the character level). As described in Section 5, our server currently supports a user community of several hundred people and processes several thousand requests on average per day without any server-related performance problems. The server typically operates continuously for two weeks between scheduled downtime for upgrades without excessive memory usage.

There are many technical details that have been addressed in the implemented server that we have not discussed here. They are layered on this basic scheme and include user password protection, groups, access control, read/write locks, change histories, and undoing changes.

4.4. Summary

In spite of the enumerated deficiencies of HTML/HTTP, we have shown that it can be used to construct an interactive editing environment whose capabilities are substantially in excess of what one might expect from the simple form and transaction based interface.

We have shown that

- 1) The stateless HTTP can be used to interact with stateful systems.
- 2) A simple mechanism (UID) and state layering provides state on a page by page, session, and/or user basis.
- 3) Layered state allows systems to go beyond distributed access to support collaboration among users.
- 4) A unique-id table can be used to represent the sparse subset of pages that a user actually touches (or creates) among the semi-infinite space of dynamically generated pages.

The mechanisms that we have described can be readily adopted by others who want to provide access to stateful collaborative applications over the web.

5. Results

In this section we discuss some of our experience in using and providing the ontology server over the web. Our general approach of providing access to research software over the web appears to be highly successful. Indeed, it has exceeded our expectations by almost every metric. We have been able to reach a wide audience in a cost effective manner and provide them with a high quality, robust, and useful tool with far higher functionality than would have been possible had we been distributing source code.

Research organizations such as KSL have limited resources to devote to maintaining, documenting, and distributing software. In general, we would rather invest our energies in adding new functionality than in porting to yet one more platform or idiosyncratic hardware and software configuration. Furthermore, the users of research software would rather invest their time and energy evaluating and using it than in downloading, configuring, and compiling. We also save them from buying the necessary software licenses and hardware platform needed to support our software. When we started work on the Ontology Server, we hoped that we would be able to impact both of these issues substantially: increase the functionality of our software while reducing the cost of maintaining and distributing software; increase the number of users by reducing their overhead. In addition, there have been numerous other benefits: users who are already familiar with their web browsers can use the ontology server right away. The constrained interface afforded through HTML has actually had positive effects on our interface design — it is much cleaner and more streamlined than it might otherwise have been. Useful context-sensitive hypertext documentation is accessible by any user through the same web interface. The HTTP server was easily extended to handle other network protocols such as the Network GFP. The modular state and session organization was easily extended to support several other network services, that are not described here (but are also available through <http://www-ksl.stanford.edu>).

Evaluating the impact that the Ontology Server has had on specific projects is rather difficult. We have not yet engaged in any formal study and the logging information that we have kept has been limited. This is partly by design and partly due to the nature of the system. We want users to feel confident about the privacy of their data and interactions with the server. This is especially important for industrial users. It is also not yet clear what sorts of information would be useful to log. Most web servers maintain a complete log of all URLs that are requested from them. This allows them to determine usage patterns such as how often specific documents are examined. In the Ontology Server, however, things are not so simple. Remember the discussion in Section 3. The URLs that go in and out of the Ontology Server encode each request with an identifier that is unique to the command, user, and session. A record of these identifiers would be useless (like recording hash keys for hash tables that have subsequently been cleared). Furthermore, the pages that users access do not come from any fixed set of files on a server. Users create, modify, and delete new objects while they work. In this effectively infinite space of objects, there is relatively little overlap between users. Thus, recording the objects that were visited might be of little interest. Although it might be interesting to record command executions, the ontology server supports over 500 distinct commands, and the current command architecture makes it difficult to draw any coherent picture of user activity from this sort of log. An important next step is to determine meaningful instrumentation that will help us evaluate the impact that the editor has on collaborative activities.

Nonetheless, we do have a base level of instrumentation that allows us to draw some inferences about the overall patterns of usage and activity.

The Ontology Server has been extremely reliable. Since its public announcement at the start of February 1995, the server has been available 99.89% of the time as is shown in Figure 6. Including scheduled downtime for hardware and software upgrades, the total downtime has been typically less than one hour per month as shown in Figure 7. This high level of reliability is essential if we expect remote users to make use of our tools and services.

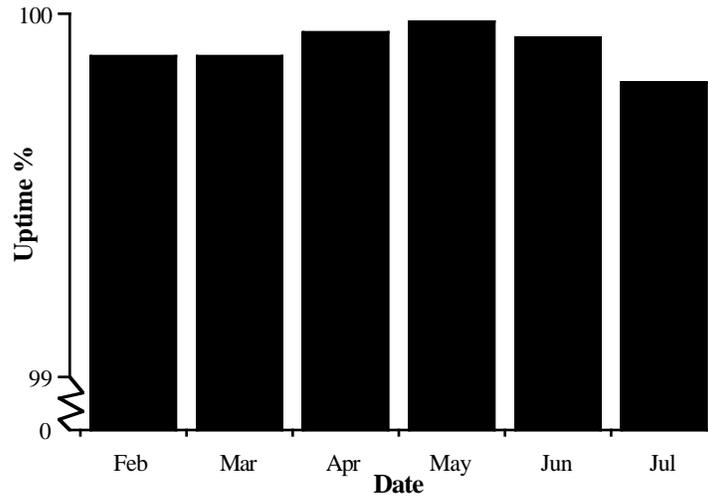


Figure 6: Percentage uptime of our server since we went public with our alpha release.

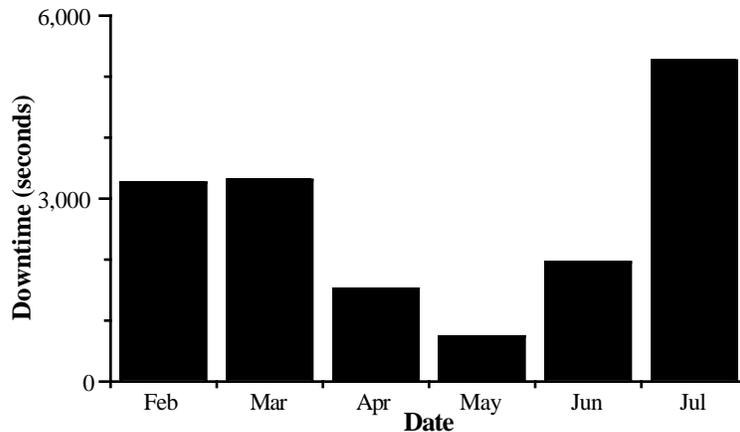


Figure 7: Server downtime, in seconds per month.

The Ontology Server has reached a wide audience with an acceptable level of continued use. Although we believe that ontologies are becoming increasingly useful and important for a number of purposes, the ontology server is still a research vehicle that is relevant to a very narrow range of people. Thus, we should not expect usage levels to approach that of very general purpose web sites such as Lycos. Furthermore, announcements of the Ontology Server have only been made to fairly focused mailing lists of researchers interested in Ontolingua, Shareable Reusable Knowledge Bases, and Qualitative Reasoning. Nonetheless, as Figure 6 indicates, the number of registered users has grown steadily. Figure 7 graphs the number of

users registered per week. The sharp increase around week 11 corresponds to the response from our early announcements to the mailing lists when we went into alpha release. It has been interesting to note that the number of users has been increasing at an almost constant rate since around week 15. We do not believe that these users have been attracted by our announcements. They have either been informed by their colleagues or have found our server through searches on one of the WWW search indices (e.g., Yahoo or Lycos — although the server is not, to our knowledge, explicitly referenced on any such index).

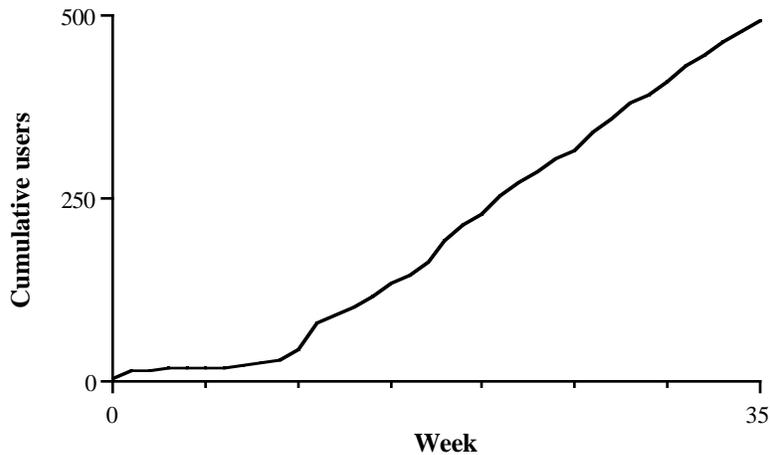


Figure 8: The number of registered users has grown steadily since the first announcements.

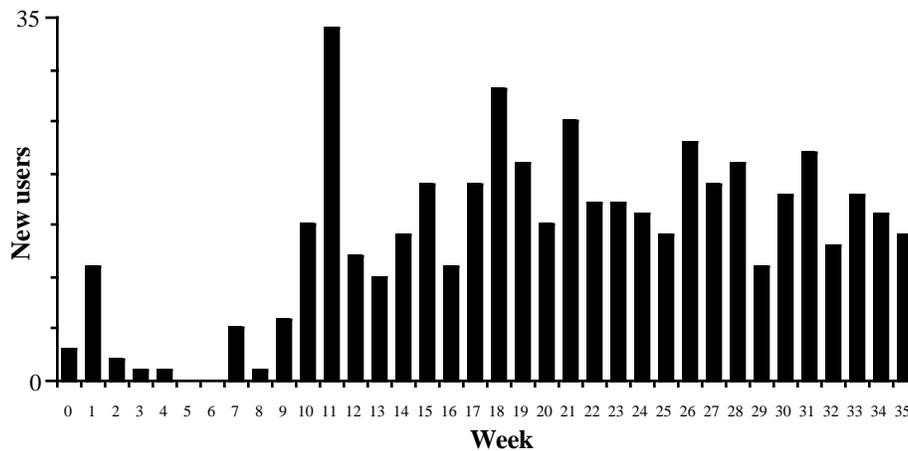


Figure 9: The number of new users registering indicates continuing interest.

Clearly, the number of users does not tell the full story . Figure 10 and Figure 11 provide some useful profile information. The first shows the distribution of users according to the total number of requests they have made. As we would expect, the largest group of users are the ones that have simply surfed by, executed a small number of requests, and moved on. Because of our general-purpose server architecture and the need for new users to register and create their accounts, a new user must execute several requests in

order to get into the ontology editor. The next few groups actually look around a bit. Examining 50 pages or so requires some investment of time and energy. These users are clearly evaluating the system and exploring some of its capabilities. This level of activity would also be consistent with colleagues or superiors looking at an ontology author's work. The next segment is issuing 50 to 500 requests. This is enough to construct sample ontologies and do student level representation exercises. These users are developing a strong sense for the system and its capabilities. Finally, there are the users who have issued thousands of requests. These are our serious users, many of whom are doing substantial ontology development. We expect that the shape of this graph will remain fairly stable over time.

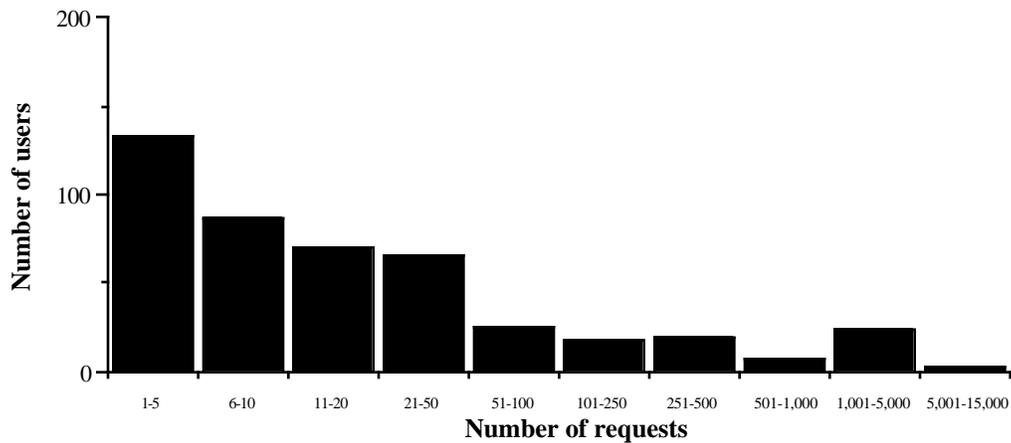


Figure 10: Users are grouped by the number of requests they make.

Figure 11 gives another view of this same information. It shows the cumulative impact on the Ontology Server's use that each of the groups in Figure 10 contributes. This graph shows that 80% of the requests are generated by 8% of the users.

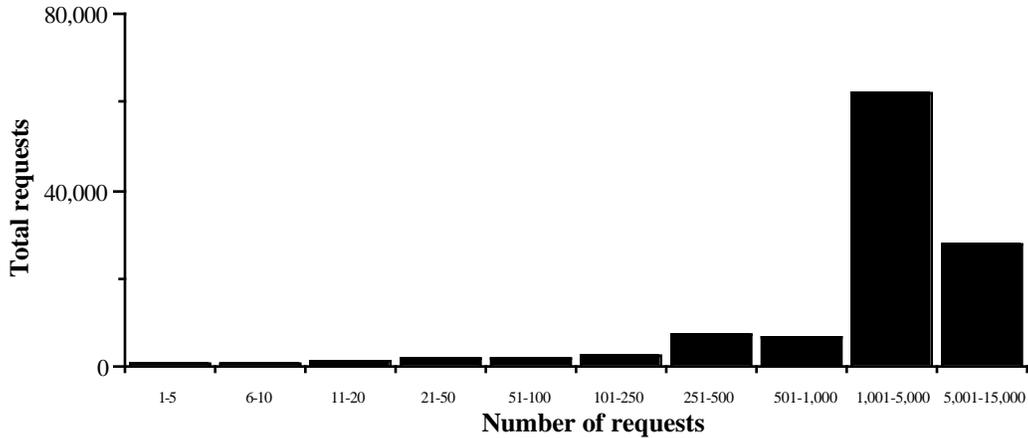


Figure 11: The total number of requests issued by the same groupings as in . These figures do not include “trivial” requests including requests for graphics or static documents .

Figure 12 shows the distribution of requests by top-level Internet domain. While the overwhelming majority of requests have been issued by U.S. educational users, there have been a substantial number issued by U.S. commercial users (COM) and active groups have been established in Europe, Great Britain, and Japan.

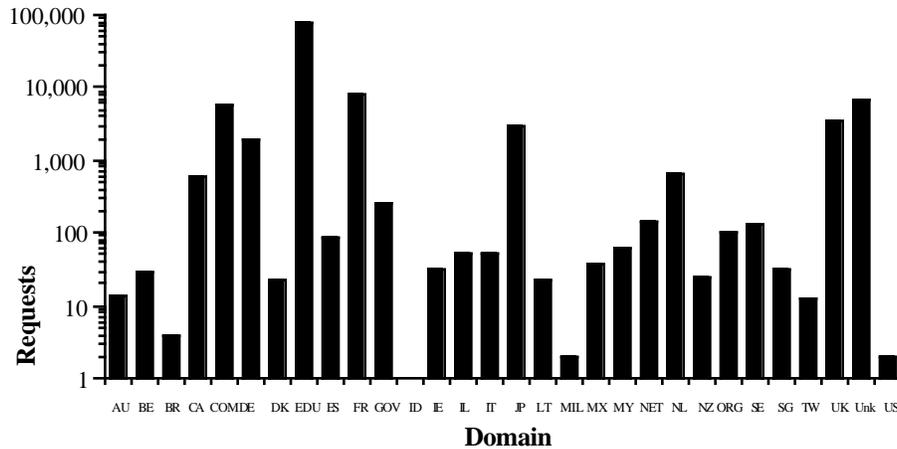


Figure 12: Number of users by Internet domain.⁴

The usage patterns are consistent with our expectations and with those generally found in the software industry. Only a small percent of the people who try the Ontology Server end up using it on a regular basis. The rest try it and move on, although some of these return later when they need the capabilities of an Ontology Server.

⁴The domain marked “Unk” accounts for all requests logged from anonymous connections. We log the requests according to the domain of the user’s email address, not the current connection location.

Evaluating the work done with the Ontology Server and its impact on collaboration is difficult. It is clear, however, that users are constructing substantial ontologies. The subject matter of the user contributed ontologies is varied. The more mature ontologies include metadata for genome database integration, satellite image metadata, enterprise integration, products and product catalogs, oscilloscopes, semiconductors, robotics, solid modeling, drugs, medical terminology, the IEEE standard 1175 for tool interconnections, and many others. Some of the larger ontologies, such as the one for medical terminology, which contains over two thousand definitions, have been imported from existing projects. Others, such as the IEEE 1175 ontology, which contains over one hundred and forty definitions, have been constructed entirely through interactions with the ontology editor.

In addition to ontology construction efforts, the Ontology Server is also being used in at least two projects to provide run-time access to ontologies.

The T-Helper medical application is an outpatient computer-based record system for patients with human immunodeficiency virus (HIV). In order to determine if patients are eligible to participate in clinical trials, it uses an ontology of drug types and specific drugs. It queries the Ontology Server to help determine if drug-related eligibility criteria are met (Gennari, Oliver, Pratt, Rice, & Musen, 1995).

The SHADE project is addressing many information system interoperability issues. The Ontology Server has been used to define several large ontologies for satellite image metadata (e.g., for the Federal Geospatial Metadata Standard, the Terramar satellite image database). Client-side tools are being developed to view the metadata and define mapping relations between concepts in them. These client-side tools extract the concept definitions from the ontologies stored on the Server.

We are tremendously pleased with the level of use that the Ontology Server has experienced. It indicates that the system is filling an important niche and that we are meeting our goal of reaching a wide audience and providing it with reliable useful tools for building and using ontologies.

6. Conclusion

In this paper, we have identified several barriers that must be overcome before ontologies can become practical, useful artifacts and we have presented several steps that we have taken towards removing these barriers.

We presented an inclusion model for ontologies that enables users to assemble new ontologies rapidly from existing ones in a repository. This model makes a clean separation between its simple formal semantics and the input/output properties of the system that uses it. The formal model handles simple inclusion, polymorphic refinement, restrictions, and circular inclusion dependencies. The input/output model yields succinct readable external representations and is transparent to users.

We described our web-based ontology editing environment that is integrated with the Ontology Server and that incorporates this inclusion model. In addition to providing individual users with a rich many-featured editing environment, this server also supports collaboration between distributed groups of users, and provides access to a growing repository of ontologies. The Ontology Server also provides a vital publishing medium for ontologies because hypertext pointers can reliably point to any document in the publicly accessible library of ontologies.

We also described an infrastructure that enables the Ontology Server to provide access to users over the world-wide web so that users can create, edit, view, and export ontologies using their own web-browsers. In order to accomplish this, layered state and sessions on top of the underlying HTTP protocol without modifying it in any way. The infrastructure that we developed can be readily adopted by other projects that want to provide support for collaborative work and wide-spread, distributed access to their applications in a cost-effective way.

Finally, we presented empirical evidence that our approach to disseminating and providing tools to promulgate the use of ontologies is effective. Our tools and the infrastructure that supports them have been extremely reliable (with a 99.9% uptime). The infrastructure scales well and currently supports several hundred users without performance problems. Users have been able to construct substantial ontologies with the web-based editor including ontologies with many hundreds of definitions. Several of these construction efforts have been collaborative with users distributed world-wide. The server architecture also supports programmatic queries from remote software agents that interrogate the server about the definitions of terms in ontologies and possibly modify them.

Constructing ontologies is a difficult, time-consuming process. The tools that we have been developing help to amortize this effort across many users and multiply the benefits across many uses.

The Ontology server is available for public use through

`http://www-ksl-svc.stanford.edu:5915/`

Acknowledgments

This research was supported by a grant from ARPA and NASA Ames Research Center (NAG 2-581), NASA Ames Research Center under contract NCC2-5337, and through CommerceNet under contract CN-1094 (TRP #F33615-94-4413).

Bibliography

- Fikes, R., Cutkosky, M., Gruber, T., & van Baalen, J. (1991). *Knowledge Sharing Technology Project Overview*. Technical Report KSL 91-71, Stanford University, Knowledge Systems Laboratory.
- Genesereth, M. R. (1990). *The Epikit Manual*. , Epistemics, Inc. Palo Alto, CA.
- Genesereth, M. R. (1991). Knowledge Interchange Format. In J. A. Allen, R. Fikes, & E. Sandewall (Eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, Cambridge, MA, pages 599-600, Morgan Kaufmann.
- Genesereth, M. R., & Fikes, R. E. (1992). *Knowledge Interchange Format, Version 3.0 Reference Manual*. Technical Report Logic-92-1, Computer Science Department, Stanford University.
- Gennari, J. H., Oliver, D. E., Pratt, W., Rice, J., & Musen, M. A. (1995). A Web-Based Architecture for a Medical Vocabulary Server. *Nineteenth Annual Symposium on Computer Applications in Medical Care*, New Orleans, LA.
- Gruber, T. R. (1992). *Ontolingua: A mechanism to support portable ontologies*. Technical Report KSL 91-66, Stanford University, Knowledge Systems Laboratory. Revision.
- Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2), 199-220.
- Karp, P. D., Myers, K., & Gruber, T. (1995). The Generic Frame Protocol. *14th International Joint Conference on Artificial Intelligence*, Montreal, Canada. Specification available at <http://www.ai.sri.com/~gfp/>.
- MacGregor, R. (1990). *LOOM Users Manual*. Working Paper ISI/WP-22, USC/Information Sciences Institute`.
- Mitchell, T. M., Allen, J., Chalasani, P., Cheng, J., Etzioni, O., Ringuette, M., & Schlimmer, J. C. (1989). *Theo: A Framework for Self-Improving Systems*. National Science Foundation, Digital Equipment Corporation, To appear as a chapter in "Architectures for Intelligence," K. VanLehn (ed.), Erlbaum, 1989.
- Mowbray, T. J., & Zahavi, R. (1995). *The ESSENTIAL CORBA: System Integration Using Distributed Objects*. John Wiley and Object Management Group.