A Generic Knowledge-Base Browser and Editor

Suzanne M. Paley, John D. Lowrance and Peter D. Karp
Artificial Intelligence Center
SRI International
333 Ravenswood Ave., EJ231
Menlo Park, CA 94025
fax: 415-859-3735, voice: 415-859-5708
{paley, lowrance, pkarp}@ai.sri.com

Abstract

The GKB Editor is a generic editor and browser of knowledge bases (KBs) and ontologies — generic in the sense that it is portable across several frame knowledge representation systems (FRSs). This generality is possible because the GKB Editor performs all KB access operations using a generic application programming interface to FRSs called the Generic Frame Protocol (GFP). To adapt the GKB Editor to a new FRS, we need only to create a GFP implementation for that FRS — a task that is usually considerably simpler than implementing a complete KB editor. The GKB Editor also contains several relatively advanced features, including three different viewers of KB relationships, incremental browsing of large graphs, KB analysis tools, extensive customizability, complex selection operations, cut-and-paste operations, and both user- and KB-specific profiles. The GKB Editor is in active use in the development of several ontologies and KBs. This paper discusses the design of the GKB Editor from a graphical user interface point of view, and describes the difficulties encountered in achieving true portability across multiple FRSs.

Introduction

In 1991 Neches et al. articulated a vision of enabling technology for knowledge sharing (Neches *et al.* 1991). That vision involved a "Chinese menu" (not to be confused with Searls' Chinese room) of interoperable, reusable components that a system designer could mix and match to construct knowledge-based systems, including knowledge representation systems, ontologies, and reasoners. The paper argued that because the de novo construction of knowledge-based systems is incredibly time consuming, future knowledge-based systems should be constructing by reusing and modifying existing components. Five years later, this vision has been realized to a limited extent; the ONTOLINGUA project has produced a library of ontologies (Gruber 1993), and the KIF project has produced a language for knowledge interchange (Genesereth & Fikes 1992), but actual examples of knowledge reuse are few. This paper reports a milestone of software reuse for knowledge-based systems by describing a knowledge-base browser and editor that is reusable across several different knowledge representation systems.

The knowledge representation (KR) community has long recognized the need for graphical knowledge-base browsing and editing tools to facilitate the development of complex knowledge bases (KBs). Frame knowledge representation systems (FRSs) from KREME (Abrett *et al.* 1987) to KEE (Kehler & Clemenson 1984) to CYCL (Lenat & Guha 1990) have included graphical KB editors to assist users in developing new KBs, and in comprehending and maintaining existing KBs. More recently, the ontology movement in AI has spurred the development of graphical ontology editors.

However, the past approach of developing KB editors that were tightly wedded to a single FRS is impractical. The substantial efforts required to create such tools become lost if the associated FRS falls into disuse. Since most FRSs share a common core functionality, a more cost-effective approach is to amortize the cost of developing a single FRS interface tool across a number of FRSs. Another benefit of this approach is that it allows a user to access KBs created using a variety of FRSs through a single graphical user interface (GUI), thus reducing the barrier for a user to interact with a new FRS. Finally, most past KB editors have implemented essentially the same functionality, presumably because each new system must be built from scratch rather than building on a previous implementation.

The GKB Editor is a generic editor and browser of KBs and ontologies — generic in the sense that it is portable across several FRSs. This generality is possible because the GKB Editor performs all KB access

and modification operations by using a generic application programming interface to FRSs called the Generic Frame Protocol (GFP) (Karp, Myers, & Gruber 1995). To adapt the GKB Editor to a new FRS, we need only to create a GFP implementation for that FRS — a task that is usually considerably simpler than implementing a complete KB editor. The GKB Editor also contains a number of relatively advanced features, such as incremental browsing of large graphs, KB analysis tools, operation over multiple selections, cut-and-paste operations, and both user- and KB-specific profiles.

The GKB Editor is in active use in the development of military-application planning KBs and ontologies for LOOM (MacGregor 1991) at several sites, including a military-transportation planning ontology. It is used daily in the development of EcoCyc (Karp *et al.* 1997), a biological KB containing more than 11,000 frames that is accessed daily via the WWW by scientists from around the world[2] (public access to EcoCyc is provided by a biology-specific GUI). The editor also works with the FRSs OCELOT (developed at SRI), SIPE–2 (Wilkins 1990), and THEO (Mitchell *et al.* 1989), and in read-only mode for ONTOLINGUA. GKB Editor development has benefited greatly from the feedback provided by the user community, incorporating a number of user suggestions into subsequent versions of the system.

This paper discusses the design of the GKB Editor from a GUI point of view, and describes the difficulties encountered in achieving true portability across multiple FRSs.

Design Goals

The GKB Editor was designed to satisfy the following criteria. (1) It must be portable across multiple FRSs. (2) Users should be shielded from as many idiosyncrasies of the underlying FRS as possible. (3) Knowledge should be presented in the most natural form, which is often graphical. There should be multiple ways to view data, depending on the user's perspective and the types of modifications they wish to make. (4) The GKB Editor should support the entire life cycle of a KB or an ontology, including design, development, maintenance, comprehension, and reuse. By "comprehension" we mean the task of understanding a new and unfamiliar KB, which is usually the first step in reuse. (5) Where appropriate, editing should be accomplished through direct pictorial manipulation. For example, if a KB is represented as a graph, then we should be able to translate an editing operation that is natural to perform on a graph to the corresponding editing operation on the KB. (6) The interface should be intuitive for the novice user to understand and manipulate, but not unduly burdensome for the expert user. Shortcuts should be available for common operations. (7) The interface should be customizable: the user should have control over both the kind and amount of information displayed, including incremental revealing, and the appearance of the displayed information.

We assert that the design requirements for an ontology editor and browser are subsumed by the design requirements for a KB editor and browser. That is, a system that adequately supports the design, development, maintenance, comprehension, and reuse of KBs will adequately support the same tasks for ontologies. The reason is that although there may be semantic differences between ontologies and KBs (Guarino & Giaretta 1995), there are no substantial symbol-level differences between the two (as Guarino points out, ontologies can include both classes and instances). Given that FRSs can serve as implementation substrates for building ontologies, the GKB Editor can therefore serve as an editor and browser of ontologies. The converse is not true: it is possible to develop ontology editors that are not adequate KB editors. For example, since KBs will generally have larger scale than ontologies, an editor that is adequate for an ontology could easily prove inadequate for a large KB.

Architecture

The GKB Editor is built upon a graph-display tool called Grasper-CL (Karp *et al.* 1994). Portability among FRSs is achieved by using the Generic Frame Protocol (GFP) to form a wrapping layer between the GKB Editor and the underlying FRSs. Figure 1 illustrates the overall architecture of the GKB Editor system. OCELOT, THEO and LOOM KBs can all be persistently stored in a relational DBMS (Karp & Paley 1995).

Our architecture naturally lends itself to distributed operation, in three possible modes. In the first mode, we insert a network connection at (A) in Figure 1 by using remote X-windows. In this mode, the GKB Editor and the FRS run in a LISP process on one machine, and the X-window graphics flow over the network to the user's workstation. In practice, this approach is slow but workable for cross-continent Internet connections; it is quite acceptable over local-area networks. Approach (B) uses a remote-procedure call implementation of GFP. With the network link at (C), the LISP process runs on the user's workstation and communicates with the DBMS server over the network using SQL. This approach faults frames across the network;

[2] See WWW URL `http://www.ai.sri.com/ecocyc/ecocyc.html`.

they are cached in the LISP process. All three of these distributed modes have been implemented and tested; our group currently uses (A) and (C), both alone and in combination.

The Generic Frame Protocol

The GFP defines a set of operations that comprise a generic API to underlying FRSs (Karp, Myers, & Gruber 1995). This generic interface layer allows an application such as the GKB Editor some independence from the idiosyncrasies of specific FRS software and enables the development of generic tools that operate on many FRSs.

Although FRS implementations have significant differences, there are enough common properties that one can describe a generic model of frame representation and specify a set of access functions for interacting with FRSs. The GFP specification defines such a generic model (with frames, classes, slots, and so forth) and consists of a library of operations (e.g., get a frame by its name, change a slot's value in a frame). An application or tool written to use these access functions can access knowledge stored in any compatible FRS. GFP implementations exist for LOOM, OCELOT, SIPE-2, THEO, and ONTOLINGUA (Gruber 1993).

Each GFP operation is defined as a CLOS generic function. The implementation of GFP for a given FRS consists of a set of CLOS methods that implement the GFP operations using calls to an FRS-specific functional interface. Since many of the GFP generic functions have default methods written in terms of other GFP methods, only a core subset of the generic functions in the specification must be implemented for a given system. The default methods can be overridden to improve efficiency or for better integration with development environments.

Although GFP necessarily imposes some common requirements on the organization of knowledge (e.g., frames, slots, and values) and semantics of some assertions (e.g., instances and subclass relationship, and inherited slot values, slot constraints), it allows for some variety in the behavior of underlying FRSs. The protocol achieves this heterogeneity by parameterizing FRSs themselves, providing an explicit model of the properties of FRSs that may vary. An application can ask for the behavior profile of an FRS, and adapt itself accordingly. For example, OCELOT supports annotations on slot values, whereas LOOM does not.[3] When annotations are present, it is desirable for the GKB Editor to display them. Before attempting this operation, however, the GKB Editor must first query an FRS's behavior profile to determine whether or not annotations are supported.

The GFP model cannot incorporate all functionalities of all FRSs. A clever translation can minimize the mismatch, however. For example, each LOOM concept has associated with it a definition, enumerating various restrictions on instances of the concept. GFP currently provides no facility for definitions (we plan to incorporate such a facility in GFP in the future, perhaps based on KRSS). However, GFP does support the notion of facets on slots (which LOOM lacks), which can be used to encode restrictions about values of that slot. In the translation between GFP and LOOM, many of the restrictions that appear in LOOM concept definitions can be converted to facet values in GFP, which can then be displayed and edited using the GKB Editor. We find that the majority of LOOM concept definitions can be translated to a facet encoding. Untranslatable definitions can be edited in an Emacs-like window in the S-expression form, using the standard LOOM concept-definition language.

There is no magic bullet whereby the GKB Editor can access idiosyncratic or newly developed FRS features that are not described by the GFP. Such features must either be integrated into the GFP as extensions, or the GKB Editor must be extended with conditionally compiled FRS-specific code.

Viewing and Editing Knowledge Bases

The GKB Editor offers four different ways to view KBs. The user can view the KB as a class-instance hierarchy graph, as a set of inter-frame relationships (this is roughly analogous to a conceptual graph representation, a semantic network, or an entity-relationship diagram), as a spreadsheet, or by examining the slot values and facets of an individual frame.[4] A set of editing operations appropriate to each view has been defined so that the displayed objects can be manipulated directly and pictorially. All editing operations translate immediately to changes in the underlying KB.

Most commonly used commands are accessible via both command menus and keystroke equivalents, and most of the time a node can be selected either by clicking on it or by typing in its name (with completion). Thus, a user can choose whether interaction is to be primarily by mouse, by keyboard, or by a combination of both. A cut-and-paste facility is also available for copying slot values, facet values, and frame names.

[3] Annotations are an extension to the frame model that our group has implemented for OCELOT. Annotations are analogous to facets, and are essentially a property list for slot values; they allow us to attach comments or citations to the literature to a particular value within a slot.

[4] Snapshots from all viewers are available on the WWW at URL `http://www.ai.sri.com/~gkb/overview.html`.

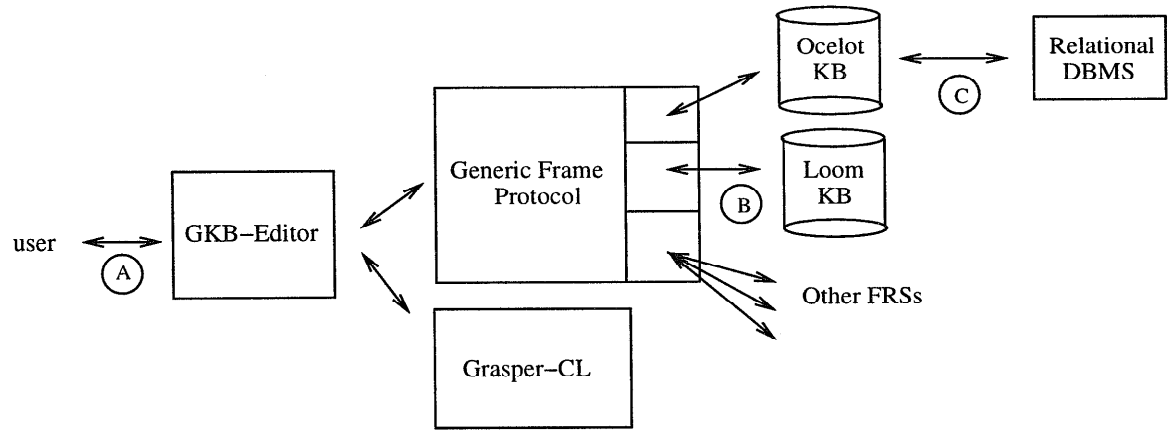

Figure 1: The architecture of the GKB Editor system. All user interaction is through the GKB Editor, which uses GFP to access KBs from a variety of FRSs, and Grasper-CL to generate and browse graphical displays.

Class-Instance Hierarchy Viewer

This viewer displays the class-instance hierarchy of a KB as a graph. Each node in the graph represents a single class or instance frame, and directed edges are drawn from a class to its subclasses and from a class to its instances. Multiple parentage is handled properly. Users can incrementally browse large hierarchies by starting at root node(s) that are either computed, or are specified by the user. The tool automatically expands the hierarchy to a preconfigured depth cutoff. If a particular node has more than a designated number of children (the breadth cutoff), the remaining children are condensed and represented by a single node. Unexpanded nodes are visually distinguished from expanded nodes. The user can browse the hierarchy by selecting, with the mouse, nodes that are to be expanded or compacted. Alternatively, the user can type in (with completion) the name of a frame that does not yet appear in the display, and, where possible, the hierarchy will be expanded out along the appropriate path until that frame is visible.

The hierarchy viewer can also be used to edit the class-instance hierarchy. Operations such as creating, deleting, and renaming frames, and altering super–subclass and class–instance links can all be accomplished with a few mouse clicks. A frame can be created either as an empty frame or as a duplicate of an existing frame. When deleting a frame, the user can choose either to delete only that frame, or to delete the entire subtree rooted at that frame. When creating or changing links, the GKB Editor checks for and disallows links that would create a cycle in the class hierarchy.

Inter-Frame Relationships Viewer

It is often useful to visualize relationships in a KB by following slot links rather than parent-child links. For example, if frame B is a value of slot X in frame A, then an edge can be drawn from node A to node B, labeled X. If we recognize that slot X represents a relationship between frames A and B, then this kind of graph is analogous to the view of a KB as a conceptual graph (although our displays do not use all the visual conventions of the conceptual graph community) or to a semantic network. This view is useful for showing relationships in the KB other than the class-instance hierarchy, for example, a Part-Of hierarchy.

The frame-relationships viewers can depict instance-level relationships and class-level relationships. The example in the preceding paragraph describes an instance-level relationship. In a class-level relationships view, an edge labeled X is drawn from class $C1$ to class $C2$ if values of slot X in instances of $C1$ are constrained to be instances of $C2$.

Like the hierarchy view, a relationships view is browsed incrementally. The user specifies a set of frames to serve as roots, and optionally a set of slots to follow (by default, all slots are followed), and the graph is expanded to the designated depth and breadth. A class-level or instance-level browse is selected automatically, depending on whether the specified roots are classes or instances. The user selects, with the mouse, nodes to be expanded or compacted. Unlike with the hierarchy view, the user cannot type in an arbitrary frame and have the browse expanded to that frame. This limitation is for efficiency reasons, because of the large number of possible paths to any frame.

Spreadsheet Viewer

The spreadsheet viewer allows frame data to be exported to NeXS, a commercial spreadsheet product for X-windows. Spreadsheets allow large volumes of data to be visualized in a very compact form, and support a variety of data analysis tools, such as X-Y plotting. User-specified instances form the rows of the spreadsheet, and user-specified slots form the columns of the spreadsheet. Users can launch the spreadsheet viewer from within the class-hierarchy viewer by clicking either on selected instances or on one or more classes; all instances of those classes are exported to the spreadsheet. Within the spreadsheet, users can both view and modify cell values, as well as adding new rows to the spreadsheet — the new rows are translated to new instances when the user terminates NeXS. A limitation of the spreadsheet viewer is its current inability to display more than one value per slot. We are investigating several approaches to overcome this limitation.

Frame-Editing Viewer

The frame-editing viewer allows the user to view and edit the contents of an individual frame. From the hierarchy viewer or the relationships viewer, the user may select a frame and display it in a frame-editing viewer. It presents the contents of a frame as a graph. Each slot name forms the root of a small tree; its children are individual slot values, and slot facets and their values. Inherited slot values are distinguished visually from local items, and cannot be edited (but they can be overridden where appropriate). The user can choose whether to display all slots and facets, filled slots and facets only, or selected slots and facets only.

In addition to duplicating, renaming, or deleting the viewed frame, the kinds of editing operations available in this viewer permit adding, deleting, replacing, and editing of slot values, facet values, and annotations. Slots themselves may be added, removed, or renamed, when classes are edited.

Customizability

The GKB Editor is highly customizable, allowing users to specify via a set of preference dialogs what should be displayed in the various viewers and how various objects should be drawn. Customizability is important if the GKB Editor is to find widespread use — the more control the user has over the appearance and behavior of the displays, the more likely that the GKB Editor will suit the user's needs.

A style dialog for the hierarchy and relationships viewers lets the user control how individual nodes are displayed. The user can specify icon and label colors, icon shape, and label font, face, and size. The user can specify a style for all frames satisfying some predicate. Any number of these styles and predicates can be specified, and frames that satisfy more than one predicate will show characteristics of each corresponding style (except where such characteristics conflict with each other). Currently, only a few predefined predicates are available, such as for identifying classes, instances, and primitive classes, for testing frames for a particular user-specified slot value, or identifying children of a class.

In these same two viewers, the user can specify slots whose values are to appear as part of the display for a frame node. This facility permits the user to see parts of each frame's interior while browsing the KB, without actually opening a frame-editing window for each frame. If the FRS provides each frame with a "pretty name", intended to be user-readable rather than machine-readable (perhaps encoded as a slot value), then the user can specify that frames should be labeled with their pretty names.

A user can define a personal preferences profile that will take effect across all KBs. In addition, because many preferences make sense only when applied to a particular KB (such as the list of browse roots or slots to be displayed), the user can save an individual profile for each KB (KB-specific preferences take precedence).

Lessons in Portability

The proof of the successful multi-FRS portability of the GKB Editor is the ongoing use of the GKB Editor to edit real-world KBs for OCELOT and LOOM. These two FRSs lie at fairly opposite ends of the FRS spectrum: LOOM is a KL-ONE descendant that supports classification. OCELOT is in the UNIT Package family of FRSs, along with KEE, CYCL, and THEO (Karp 1992). It does not perform classification — all class–subclass and class–instance relationships are specified by the user.. It does support facets (both built-in facets such as value-type and cardinality, and user-specified facets) and annotations, which LOOM does not. OCELOT supports multiple parentage, procedural attachment, constraints on slot values, and runtime schema alterations (e.g., changes to class definitions for a loaded KB). OCELOT KBs can be stored in files or in a database back-end (Karp & Paley 1995).

Here we summarize the limitations that exist and the difficulties that were encountered in making the GKB Editor truly generic. Limitations: (1) GFP supports neither contexts nor complex inheritance relationships among KBs; the GKB Editor does not recognize these constructs. (2) The section on The Generic Frame Protocol discusses the partial mapping we have defined between facets and the LOOM concept definition lan-

guage. Although this approach works in practice, use of a common concept-definition language would make multiple description-logic systems look more uniform to the user. (3) Any FRS will have idiosyncrasies that fall outside the GFP model. For example, LOOM provides three alternative implementations of instance frames, which are not supported by GFP. We extended the GKB Editor to recognize this construct.

FRSs vary in the degree of dynamic schema alteration (changes in class definitions) they allow; LOOM and OCELOT are flexible in this regard, whereas CLASSIC, for example, is not. The GKB Editor cannot provide an FRS with functionality that the FRS lacks.

Related Work

Many graphical browsers and editors have been built for individual FRSs. We have built on ideas from several of these systems. KnEd (Eilerts 1994) offers two types of viewers, a graphical hierarchy viewer analogous to both our hierarchy and relationships viewers (although KnEd does not support editing operations) and a textual viewer for browsing and editing individual frames and slots. The user interface for the University of Ottawa's CODE4 knowledge management system (Skuce & Lethbridge 1995) offers a spreadsheet view as well as textual outline and semantic net views of a KB. "Masks" let the user control what KB elements are visible and, to a limited extent, how they should be displayed. The HITS Editor (Terveen & Wroblewski 1990) supports browsing and editing of the CYC (Lenat & Guha 1990) KB. It defines user *perspectives* on a per-class basis to determine what information should be displayed, and builds checklists for data entry tasks.

Protege-II is a powerful suite of knowledge-acquisition tools (Eriksson *et al.* 1994). One tool supports ontology editing; a second tool accepts an ontology as input, and produces as output a specification of a forms-based editor for instance frames within that ontology. Protege-II includes a relationships viewer. Protege-II and the GKB Editor embody different approaches for visualizing frames. We use a graph visualization of frames rather than a forms-based visualization because the forms approach is problematic for encoding slots with multiple values (it is often not clear how many blank form elements to allocate for new values of a multivalued slot), and for representing facets and annotations — the graph visualization makes the relationships among a slot, its facets, its values, and their annotations very evident. In addition, although Protege-II gets significant mileage from decoupling the editing of ontologies (classes) and instances, there are two problems with this approach: first, during the KB development process these two operations are often tightly coupled — a user may alternate between editing of classes and instances frequently, and may prefer to avoid the process of generating a new user interface for instance editing after every class change; second, some modifications to classes actually demand immediate modifications to instances of those classes, for example, when deleting a slot from a class, the GKB Editor will automatically delete all occurrences of that slot from instances of the class (with user confirmation). It is not clear how Protege-II can modify instances in response to class changes.

The preceding tools operate with only one FRS.

Stanford's Ontology Editor (Farquhar *et al.* 1995) is a browser and editor for shared ontologies, encoded using the ONTOLINGUA language (Gruber 1993). Users access the interface using the WWW. Currently, the Ontology Server operates only on ONTOLINGUA ontologies. Because it is implemented using GFP, and because some translators between ONTOLINGUA and other representations either exist or are under development, the Ontology Server could in principle be used to browse and edit KBs for a variety of KR systems. The WWW implementation of the Ontology Editor is both its biggest advantage and its biggest drawback. The advantage is the easy accessibility; the drawbacks result from the many limitations of the HTTP protocol: most information is presented in textual form, rather than graphically; displays cannot be updated incrementally, as they can in the GKB Editor — the only operation within HTTP is to send an entire new page, which can be slow; the lack of state in HTTP limits the style of user interaction that can occur, as does the few mouse events supported by HTTP. Use of Java would overcome many of these limitations of HTTP.

Performance and Availability

The performance of the GKB Editor is quite acceptable on a SPARC-20 workstation (now an outdated machine) with 48 MB of memory; graphics operations execute fast, and the incremental browsing facilities provide fast browsing of the EcoCyc biology KB, which contains more than 11,000 frames.

The GKB Editor is available under license from SRI International at no charge.[5]

Acknowledgments

We thank Jon Doyle, Bob MacGregor, and Tom Russ for comments and suggestions for the GKB Editor. Ife Olowe provided programming assistance. This

[5]See WWW URL `http://www.ai.sri.com/~gkb/`.

work was supported by DARPA Contract F30602-94-C-0263, and by grant R29-LM-05413-01A1 from the National Institutes of Health. The contents of this article are solely the responsibility of the authors and do not necessarily represent the official views of the Advanced Research Projects Agency or of the NIH.

References

Abrett, G.; Burstein, M.; Gunsbenan, J.; and Polanyi, L. 1987. KREME: A user's introduction. Technical Report 6508, BBN Laboratories Inc., Cambridge, MA.

Eilerts, E. 1994. KnEd, an interface for a frame-based knowledge representation system. Master's thesis, University of Texas at Austin.

Eriksson, H.; Puerta, A. R.; Gennari, J. H.; Rothenfluh, T. E.; Tu, S. W.; and Musen., M. A. 1994. Custom-Tailored Development Tools for Knowledge-Based Systems. Technical Report KSL-94-67, Stanford University Knowledge Systems Laboratory.

Farquhar, A.; Fikes, R.; Pratt, W.; and Rice, J. 1995. Collaborative ontology construction for information integration. Technical Report KSL-95-63, Stanford University, Knowledge Systems Laboratory.

Genesereth, M. R., and Fikes, R. E. 1992. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford University.

Gruber, T. 1993. A translation approach to portable ontology specifications. *Knowledge Acquisition* 5(2):199–220. URL for Ontolingua is http://www-ksl.stanford.edu/knowledge-sharing/ontolingua/README.html.

Guarino, N., and Giaretta, P. 1995. Ontologies and knowledge bases towards a terminological clarification. In *Towards very large knowledge bases.* Amsterdam: IOS Press. 25–32.

Karp, P., and Paley, S. 1995. Knowledge representation in the large. In *Proc of the 1995 International Joint Conference on Artificial Intelligence*, 751–758. See also WWW URL `ftp://ftp.ai.sri.com/pub/papers/karp-perkobj95.ps.Z`.

Karp, P.; Lowrance, J.; Strat, T.; and Wilkins, D. 1994. The Grasper-CL graph management system. *LISP and Symbolic Computation* 7:245–282. See also SRI Artificial Intelligence Center Technical Report 521.

Karp, P.; Riley, M.; Paley, S.; Pellegrini-Toole, A.; and Krummenacker, M. 1997. EcoCyc: Electronic encyclopedia of *E. coli* genes and metabolism. *Nuc. Acids Res.* 25(1).

Karp, P.; Myers, K.; and Gruber, T. 1995. The generic frame protocol. In *Proc of the 1995 International Joint Conference on Artificial Intelligence*, 768–774. See also WWW URL `ftp://ftp.ai.sri.com/pub/papers/karp-gfp95.ps.Z`.

Karp, P. 1992. The design space of frame knowledge representation systems. Technical Report 520, SRI International AI Center. URL `ftp://www.ai.sri.com/pub/papers/karp-freview.ps.Z`.

Kehler, T., and Clemenson, G. 1984. KEE the knowledge engineering environment for industry. *Systems And Software* 3(1):212–224.

Lenat, D., and Guha, R. 1990. *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project.* Addison-Wesley.

MacGregor, R. 1991. The evolving technology of classification-based knowledge representation systems. In Sowa, J., ed., *Principles of semantic networks.* Morgan Kaufmann Publishers. 385–400.

Mitchell, T.; Allen, J.; Chalasani, P.; Cheng, J.; Etzioni, E.; Ringuette, M.; and Schlimmer, J. 1989. Theo: A framework for self-improving systems. In *Architectures for Intelligence.* Erlbaum.

Neches, R.; Fikes, R.; Finin, T.; Gruber, T.; Patil, R.; Senator, T.; and Swartout, W. 1991. Enabling technology for knowledge sharing. *AI Magazine* 12(3):36–56.

Skuce, D., and Lethbridge, T. 1995. CODE4: A unified system for managing conceptual knowledge. *International Journal of Human-Computer Studies.*

Terveen, L., and Wroblewski, D. 1990. A collaborative interface for editing large knowledge bases. In *Proc of the Eighth National Conference on Artificial Intelligence.*

Wilkins, D. 1990. Can AI planners solve practical problems? *Computational Intelligence* 6(4):232–246.