RL-TR-97-114
Final Technical Report
October 1997

A GENERIC KNOWLEDGE-BASE BROWSER AND EDITOR

SRI International

John D. Lowrance, Suzanne M. Paley, Peter D. Karp,
and Ife Olowe

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-114 has been reviewed and is approved for publication.

APPROVED: Craig S Anken

CRAIG S. ANKEN
Project Engineer

FOR THE DIRECTOR: John A Graniero

JOHN A. GRANIERO, Chief Scientist
Command, Control, & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3CA, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1997	Final Sep 94 - Jan 97

4. TITLE AND SUBTITLE

A GENERIC KNOWLEDGE-BASE BROWSER AND EDITOR

5. FUNDING NUMBERS

C - F30602-94-C-0263
PE - 62702F
PR - 5581
TA - 27
WU - 81

6. AUTHOR(S)

John D. Lowrance, Suzanne M. Paley, Peter D. Karp, and Ife Olowe

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

SRI International
333 Ravenswood Avenue
Menlo Park CA 94025

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/C3CA
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-97-114

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Craig S. Anken/C3CA/(315) 330-4833

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Generic Knowledge-Base Editor (EKG-Editor) is a generic editor and browser of KBs and ontologies - generic in the sense that it is portable across several FRSs. This generally is possible because the GKB-Editor performs all KB access and modification operations by using a generic application programming interface (API) 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, 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.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Knowledge-Base, Editor, Ontologies	72
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

Preface

The Generic Knowledge-Base Editor (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 four 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 report discusses the design methodology behind the GKB-Editor and describes how to use the GKB-Editor. As such, this report constitutes both the final technical report and the user's manual. The most recent version of the user's manual and other information about the GKB-Editor can be found on the World Wide Web at `http://www.ai.sri.com/~gkb/`.

Chapter 1

Introduction

In 1991, Neches et al. articulated a vision of enabling technology for knowledge sharing [16]. That vision involved a "Chinese menu" (not to be confused with Searle's 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 constructed 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 [6], and the KIF project has produced a language for knowledge interchange [5], but actual examples of knowledge reuse are few. Here we report the achievement of a milestone in software reuse for knowledge-based systems, the development of 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 [1] to KEE [12] to CycL [13] 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 artificial intelligence 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 Generic Knowledge-Base Editor (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 (API) to FRSs called the Generic Frame Protocol (GFP) [10]. 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, 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 [14] at several sites, including a military-transportation planning ontology. It is used daily in the development of EcoCyc, a biological KB containing more than 11,000 frames that is accessed daily via the WWW by scientists from around the world [8] (public access to EcoCyc is provided by a biology-specific GUI). The editor also works with the FRSs Ocelot (developed at SRI), SIPE-2 [20], and Theo [15], and in read-only mode for Ontolingua. GKB-Editor development has benefited greatly from the feedback provided by the user community, incorporating user suggestions into subsequent versions of the system.

Chapter 2

Design Goals

The GKB-Editor was designed to satisfy the following criteria.

- It must be portable across multiple FRSs.
- Users should be shielded from as many idiosyncrasies of the underlying FRS as possible.
- Knowledge should be presented in the most natural form, which is often graphical. There should be multiple ways to view data, depending on the users perspectives and the types of modification they want to make.
- 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.
- 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.
- 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.
- 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 [7], 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.

Chapter 3

Architecture

The GKB-Editor is built upon a graph-display tool called Grasper-CL [9]. 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 3.1 illustrates the overall architecture of the GKB-Editor system. Loom KBs can be persistently stored in a relational DBMS [11].

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 3.1 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; they are cached in the Lisp process. Our group currently uses (A) and (C), both alone and in combination.

3.1 The Generic Frame Protocol

The GFP defines a set of operations that constitute a generic API to underlying FRSs [10]. 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

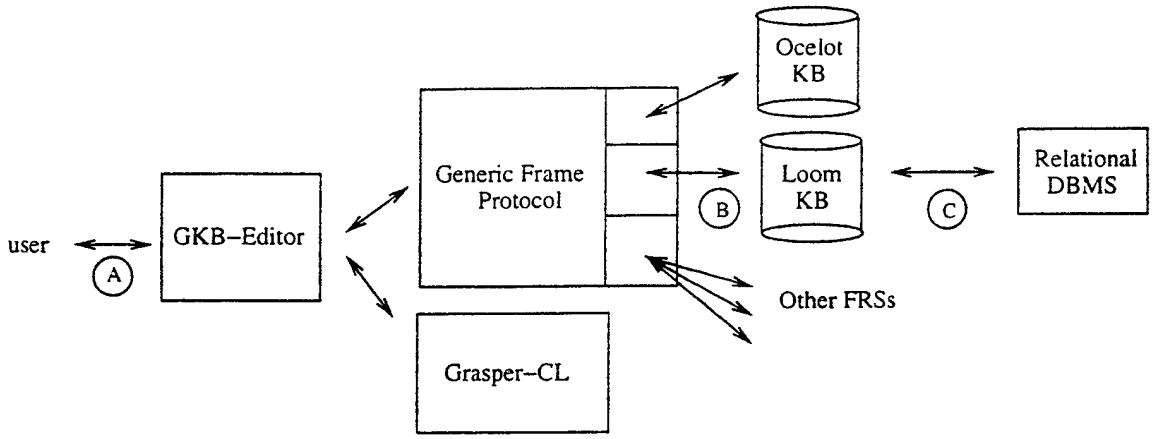


Figure 3.1: The architecture of the GKB-Editor system. All user interaction is through the GKB-Editor, which uses GFP to access KBs from a variety of FRSs, and Grasper-CL to generate and browse graphical displays.

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, SIPE-2, Theo, and Ontolingua [6].

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, Theo supports annotations on slot values, whereas Loom does not.[1] 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.

[1]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 given slot value.

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. For details on the implementation of GFP for Loom, see Appendix B.

3.2 Grasper-CL

Grasper-CL is a system for viewing and manipulating graph-structured information, and for building graph-based user interfaces for application programs. Grasper-CL defines a graph abstract datatype plus a comprehensive and novel language of operations on that datatype. This datatype includes labeled subgraphs consisting of labeled nodes connected by directed labeled edges. Grasper-CL includes procedures for graph construction, modification, and queries as well as a menu-driven, interactive, layout and drawing package that allows graphs to be constructed, modified, and viewed through direct pictorial manipulation. Nodes can appear in various shapes, including simple geometric figures (e.g., circles, rectangles, diamonds) and user-defined icons; arcs can appear in various forms, including piecewise linear or arbitrarily curved arrows between nodes. User-definable actions are associated with every graphical object, providing complete control of mouse interactions with graphs. The Grasper-CL system consists of several different components: a core procedure library for programmatically manipulating the graph abstract datatype, a graph-display module for producing drawings of graphs, a graph editor that allows users to interactively draw and edit arbitrary graphs, and a suite of automatic graph-layout algorithms. Grasper-CL has proven to be an extremely flexible support system for work in expert systems and related AI topics, where it has been customized and used as a specialized knowledge browser and editor for selected systems.

The architecture of Grasper-CL is shown in Figure 3.2. The Grasper-CL core provides a library of functions for manipulating a graph abstract datatype. These func-

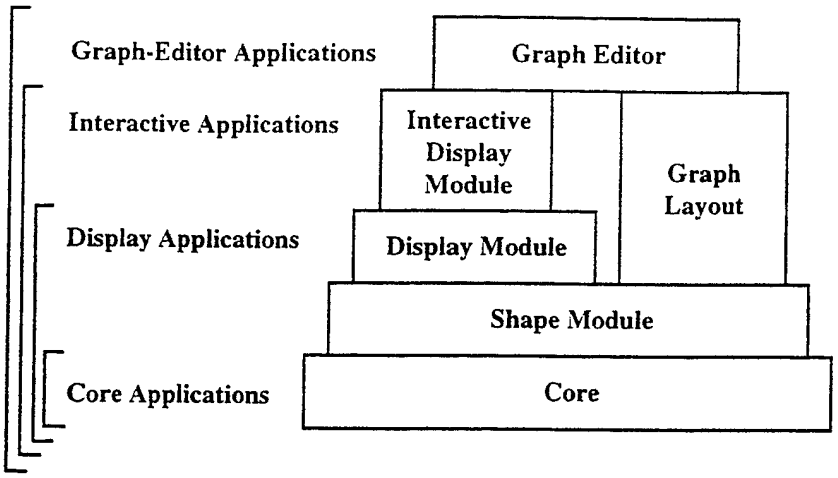


Figure 3.2: The Architecture of Grasper-CL. Grasper-CL's architecture is open and layered. Core applications use only Grasper-CL's core database function. Display applications utilize the shape and display modules, in addition to the core, to draw depictions of graph data structures. Interactive applications utilize the interactive display module and graph layout algorithms, in addition to the display, shape, and core components, to support direct user interaction with the graphical depictions of graph data structures. Graph-Editor applications utilize all of Grasper-CL's components to develop interactive graph editing systems with the look and feel of Grasper-CL's graph editor.

tions perform no drawing operations; they provide a memory-resident graph database facility in which nodes and edges can be created, destroyed, and modified, and relationships among nodes and edges can be queried. A graph can also be written to and read from a disk file.

The shape module manages information about the shapes and locations of nodes and edges, and performs shape-related calculations, such as determining the bounding box of a given node.

Given a set of nodes and edges defined using the Grasper-CL core, and shape and location information defined by the shape module, the display module draws or erases a graphical rendition of a graph, or of individual nodes and edges. Node positions might have been determined interactively by the user, or programmatically by either an application-specific layout algorithm, or by a member of the Grasper-CL suite of graph layout algorithms. Grasper-CL provides several different layout algorithms that produce completely different styles of graph layout, such as a tree or a rectangle, that can be composed to create hierarchical layouts.

The graph editor is a menu-driven Grasper-CL application that allows users to interactively draw and edit graphs. It includes operations for interactively creating, renaming, deleting, copying, or altering the shape of nodes and edges. The interactive

display module includes the functions that implement the operations within the graph editor. These functions provide a high-level substrate for building interfaces in which users also perform interactive graph editing, but with added application semantics.

Grasper-CL has an open architecture because every level in Figure 3.2 is documented and available to applications. The left side of Figure 3.2 shows different classes of applications that make use of different subsets of the Grasper-CL system. Core applications have no graphics component; they simply use the Grasper-CL core procedures to manipulate instances of the graph abstract datatype. For example, imagine that we want to program a solution to the traveling salesman problem. We could use the core procedures to create a graph whose nodes and edges represent cities and highways, to store intercity distance information at each edge, and to query these relationships during the optimization process.

Display applications use the Grasper-CL display module to draw a Grasper-CL graph. For example, we might want to display the final solution computed by our traveling salesman algorithm in a single window. Display applications support only the most primitive user interaction with a graph, such as clicking on a node to perform some action. For example, we might allow the user to left click on an edge to query the distance between two cities, and to modify the computed solution by middle clicking on an edge to delete it. The entire graph is displayed at once in a scrollable window.

Interactive applications use the Grasper-CL display module to display graphs, and they use the interactive display module procedures to support direct user interaction with the graph. Interactive applications include multiple display panes, one or more of which contains the display of a graph. Other display panes typically contain a menu of commands that can be selected for application to the graph. For the traveling salesman graph, one menu command might prompt the user to click on a city, and then display the population of the city. Another menu command might prompt the user to create a new city by clicking on the background of the graph pane, and then draw the new city at that location.

Graph-Editor applications are a restricted type of interactive applications: restricted in the sense that the style of user interaction shares much in common with the style of the Grasper-CL graph editor. Such applications are faster to program than interactive applications because of the potential for reusing code within the graph editor. For example, imagine that we want to provide the user of the traveling salesman system with the ability to edit a solution graph in arbitrary ways that incorporate the semantics of the application (for example, by moving cities, adding cities, copying cities, changing the drawn shape of cities). It is easy to define such commands by using the procedures from which the graph editor is implemented.

Chapter 4

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.[1] 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.

4.1 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 a root node(s) that is either computed, or 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

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

are condensed and represented by a single node. Unexpanded nodes are visually distinguished from expanded nodes. Figure 4.1 illustrates a sample hierarchy graph display. 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.

4.2 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 $C2$. An example illustrating the difference between these two viewers is shown in Figure 4.2.

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. An example instance-level relationships browse is shown in Figure 4.3. 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.

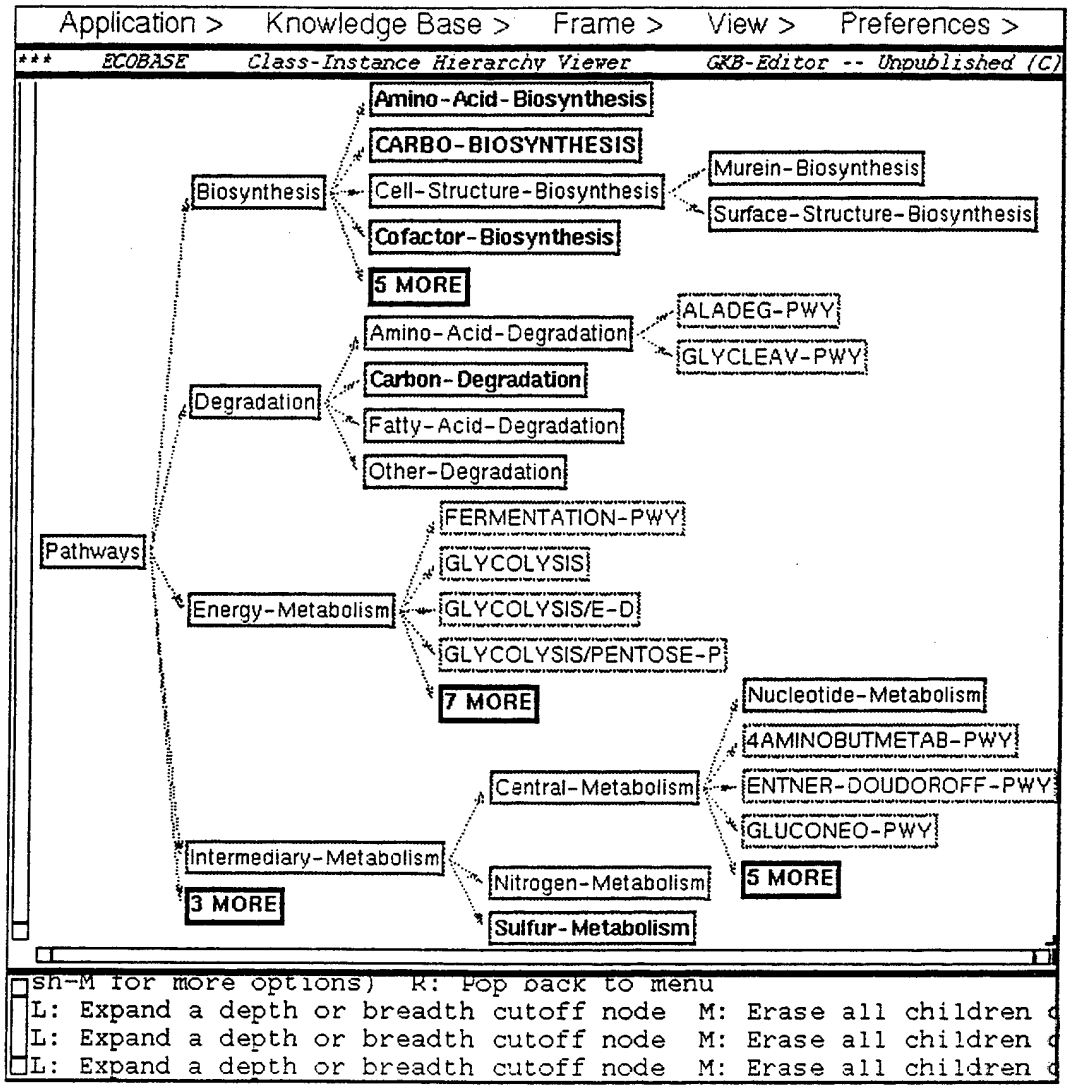


Figure 4.1: A sample class and instance hierarchy. Nodes whose labels are in boldface can be further expanded. Class frames have a solid border; instance frames have a gray border. Class names shown in bold can be expanded to show additional subclasses or instances. The "N More" nodes are "breadth cutoff nodes" that can also be expanded to show additional classes or instances.

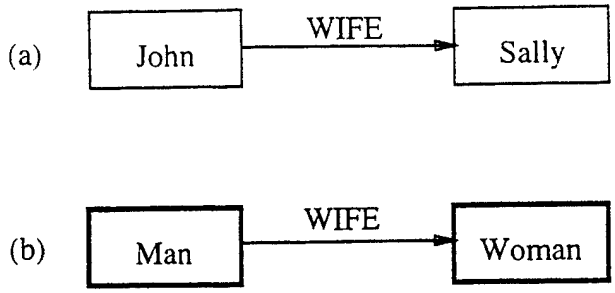


Figure 4.2: Examples of instance-level and class-level relationship graphs. (a) Instance-level relationship: the instance Sally is a value of the slot Wife in the instance John. (b) Class-level relationship: values of the slot Wife in instances of the class Man must be instances of the class Woman.

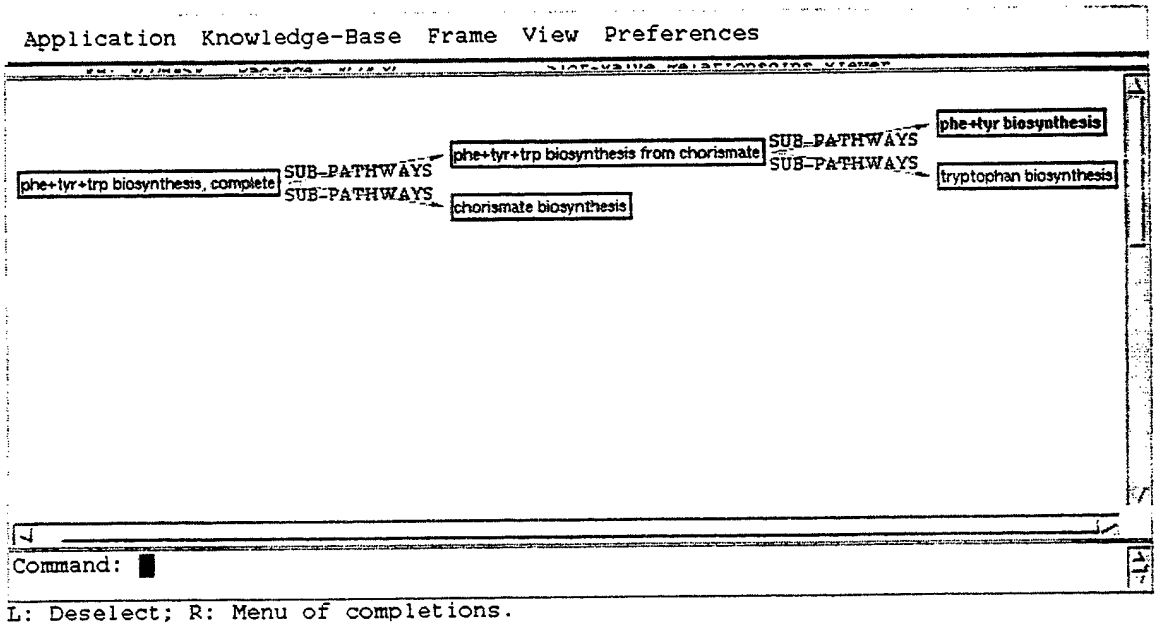
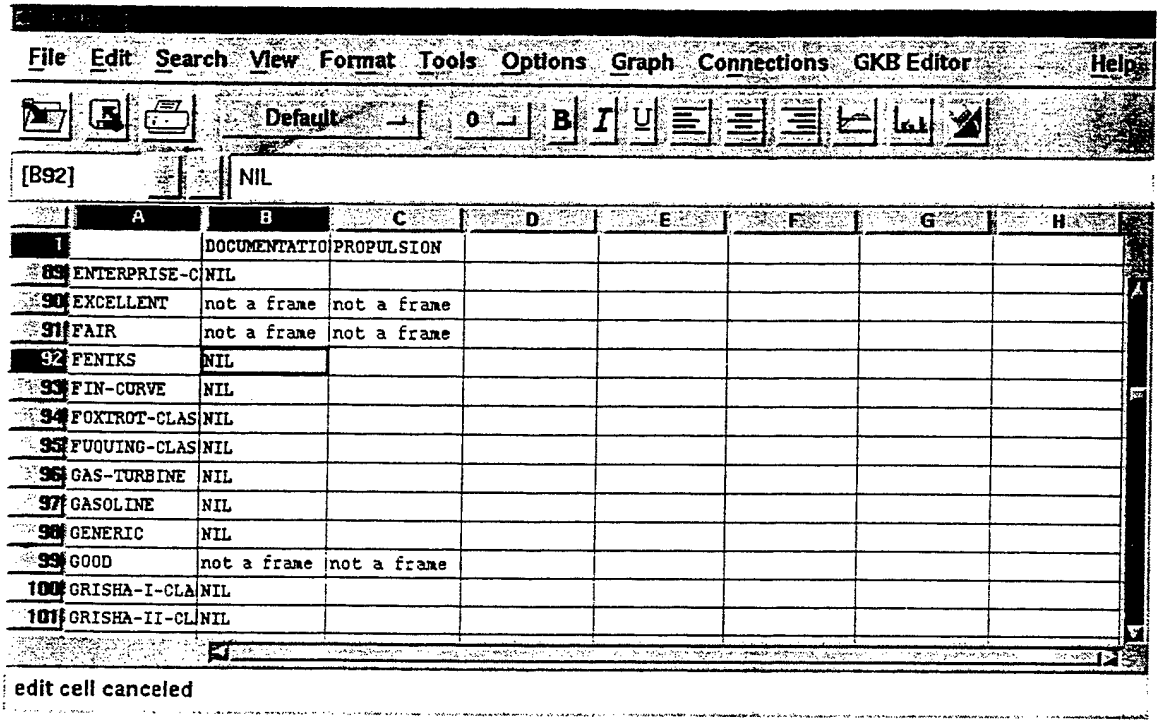


Figure 4.3: A sample instance-level relationships browse. Nodes with boldface labels can be further expanded. Double-headed arrows show bidirectional links, which are encoded by two slots that are mutual inverses.



	A	B	C	D	E	F	G	H
1		DOCUMENTATIO	PROPULSION					
89	ENTERPRISE-C	NIL						
90	EXCELLENT	not a frame	not a frame					
91	FAIR	not a frame	not a frame					
92	FENIKS	NIL						
93	FIN-CURVE	NIL						
94	FOXTROT-CLAS	NIL						
95	FUQUING-CLAS	NIL						
96	GAS-TURBINE	NIL						
97	GASOLINE	NIL						
98	GENERIC	NIL						
99	GOOD	not a frame	not a frame					
100	GRISHA-I-CLA	NIL						
101	GRISHA-II-CL	NIL						

Figure 4.4: A sample spreadsheet view. Each row within the spreadsheet corresponds to a class. The left-most column lists the name of the instance, while the remaining columns list slot values.

4.3 Spreadsheet Viewer

The spreadsheet viewer (Figure 4.4) 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 classes form the columns of the spreadsheet. Users can launch the spreadsheet viewer from within the class-hierarchy viewer by clicking 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.

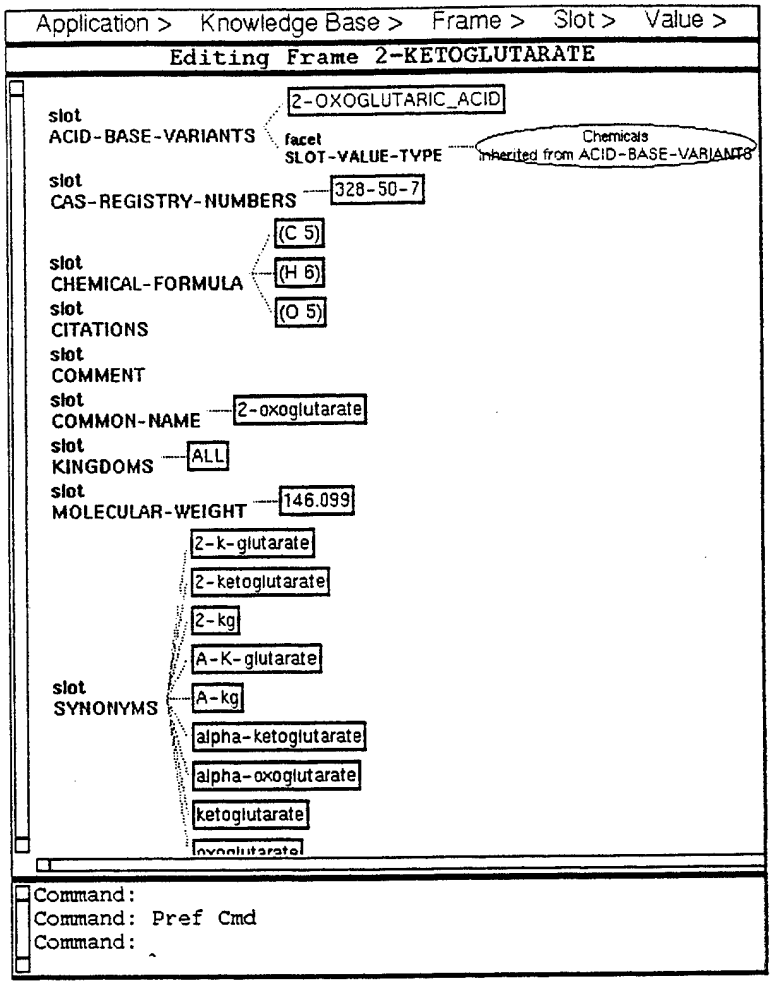


Figure 4.5: A sample frame-editing view for a chemical-compound frame. The slot-value-type facet on the acid-base-variants slot indicates that all values of this slot must be instances of the class Chemicals.

4.4 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. An example of a frame-editing view is shown in Figure 4.5.

In addition to duplicating, renaming, or deleting the viewed frame, the kinds

of editing operations available in this view 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.

Chapter 5

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 all children of a particular 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).

Chapter 6

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 conceptually a descendant of the Unit Package that supports facets and annotations, which Loom does not.

Here we summarize the limitations that exist and the difficulties that were encountered in making the GKB-Editor truly generic.

- GFP supports neither contexts nor complex inheritance relationships among KBs, and therefore the GKB-Editor does not recognize these constructs.
- Section 3.1 discusses the partial mapping we have defined between facets and the Loom concept definition language. 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.
- Any FRS has 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 in a Loom-specific fashion to recognize this construct.

FRSs vary in the degree of dynamic schema alteration (changes in class definitions) they allow: Loom and Ocelot are quite flexible in this regard, whereas Classic, for example, is not. Generally speaking, the GKB-Editor cannot provide an FRS with functionality that the FRS lacks.

Chapter 7

Related Work

Many graphical browsers and editors have been built for individual FRSs. We have built on ideas from several of these systems. KnEd [2] is a graphical editor and browser for the KM [17] FRS from the University of Texas at Austin. It offers two types of viewer, 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 [18] offers a spreadsheet view as well as textual outline and semantic net views of a knowledge base, and allows both browsing and editing. "Masks" let the user control what KB elements are visible and, to a limited extent, how they should be displayed. A feedback panel describes changes in response to editing commands, and lists options in the case of constraint violations. The HITS Knowledge Editor [19] supports browsing and editing of the CYC [13] 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 [3]. 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 complexities such as 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 frequently alternate between editing of classes and instances, 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.

All the preceding tools operate with only a single FRS.

Stanford's Ontology Editor [4] is a browser and editor for shared ontologies, encoded using the Ontolingua language [6]. Users access the interface by 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 of the server: its 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.

Chapter 8

Using the GKB-Editor

8.1 Obtaining and Installing the GKB-Editor

The GKB-Editor can be obtained from SRI International via the World Wide Web. For instructions see the GKB-Editor home page at `http://www.ai.sri.com/~gkb/`. The associated pages describe how to obtain, install, and run the GKB-Editor. Since these pages are updated with each new release of the GKB-Editor, they are the best source for up-to-date information.

8.2 Invoking the GKB-Editor

After you have loaded a Lisp image containing the GKB-Editor and one of the supported frame systems, type `(user:run-gkb-editor)` in the Lisp Listener to start the GKB-Editor. A large window appears containing a copyright notice and divided vertically into five parts:

- **menu bar**
  This is a list of pull-down (or in CLIM 1.1, pop-up) command menus. The individual commands are each described in the appropriate sections of this manual.

- **status line**
  This line contains several important pieces of information. From left to right, it lists the name and package of the current KB, and the type of viewer in operation (if you have just started up, this will be the Class-Instance Hierarchy Viewer). In addition, if the KB has been modified since it was last saved, this is indicated by three asterisks at the far left.

- **graphing pane**
  This large pane containing the copyright notice contains graphical output produced by the GKB-Editor. Items displayed here are sensitive to pointer clicks.

- **text pane**
  This small pane is for textual input and output. Warning messages from the underlying FRS appears here, as well as feedback and instructions from GKB-Editor commands. When asked to select a frame, you can type in its name here instead of clicking in the graphing pane.

- **pointer documentation line**
  When the pointer passes over a sensitive item, the operations that correspond to left, middle, and right clicks are listed here. Holding down, for example, the shift key shows operations that correspond to the various mouse clicks with the shift key held down.

If, when you start, a KB has already been selected (either because you used the GKB-Editor to select it last time, or because you opened it using the GFP), you can begin browsing the KB immediately. Otherwise, you must open and select a KB before you can begin browsing or editing. If no KB is currently open, you will see the message *Create or Open a KB* in the status line where you would normally see the name and package of the KB.

The first time you open a KB, you must use the Open or New command, which pops up a dialog, prompting you for a filename and other information about the KB. This information is saved persistently, so the next time you want to open the same KB, you need only select its name from the list of KBs provided when you invoke the Open command.

8.3 Common Features of the GKB-Editor Viewers

The GKB-Editor features four types of viewer, the class hierarchy viewer, the frame relationships viewer, the frame-editing viewer, and the spreadsheet viewer. The original application window that appears when you first start up the GKB-Editor is the class hierarchy viewer. From this window, you can pop up any of the other viewers. However, many of the application and knowledge base operations are accessible only from this original application window, so you must eventually return to it. Descriptions and examples of each of the available viewers are given in the appropriate sections. The remainder of this section describes things that are common to all three viewers.

8.3.1 Lisp Packages

When opening a KB for the first time, you are asked to enter (or verify) its package. The package name also appears with the KB name on the status line for the hierarchy and relationships viewers. This section is provided for users who are unfamiliar with the Lisp package system. Experienced Lisp users will recognize this section as greatly oversimplifying the Lisp package system.

A Lisp package defines a namespace. Within a single package, no two symbols can have the same name. Since frame names are Lisp symbols, no two frames can have the same name (in some frame systems, the same name can be used for both a frame and a slot, but because confusion may arise, this practice is not recommended). Lisp has a notion of the *current package*, such that by default, any symbol you reference is assumed to be in that package. You can still reference symbols in other packages, but you must preface the symbol name with the package name and `::`.

For example, suppose there are two packages `PKG-X` and `PKG-Y`, and assume there is a symbol `A` present in both packages. If my current package is `PKG-X` and I refer to `A`, then I am referring to the symbol `A` in `PKG-X`. To refer to the symbol `A` in `PKG-Y`, I must write `PKG-Y::A`.

When a KB is created or opened, it is important to know which package all its symbols should be in by default. When you first start up Lisp, you are in some default package, usually `CL-USER`. When you open or select a KB, that KB's package becomes the current package. This way, when you are asked to enter a frame name, that frame name by default becomes a symbol in the KB's package. (You can still reference symbols in other packages, but you must use the `::` notation). Youmight want to put each KB in its own package so that you can have multiple KBs open at once without their frame names conflicting. When you type in a package name while opening or creating a KB, and the package does not yet exist, it is created.

8.3.2 Frame Selection

Many of the menu commands are designed to operate on a particular frame or set of frames. The GKB-Editor allows you either to select the frame(s) first and then select the operation, or to select an operation, which then prompts you for the frame(s). However, if you prefer the latter mode of operation, you must make sure that there is no current selection when you enter a command, or else the command may be performed on the wrong frame.

When no command is currently active, left clicking on a node causes that frame to be selected. To indicate this, the node is shown in reverse video. Shift-left clicking on a node adds it to, or removes it from, the selection, allowing you to select multiple frames for a single operation. Left clicking on the background cancels all current selections. In addition, there are three commands accessible from the top-level **Frame**

Menu in the hierarchy and relationships viewers, **Select All**, **Select Descendants**, and **Select Ancestors**, that will select a whole group of frames. After a selection is made, an operation can be selected from the command menus. If the operation is one that can operate on multiple frames, the operation is performed on the entire selection. If the operation requires a single frame, and more than one frame has been selected, you are offered a menu of selected frames from which to choose. If, for example, the operation is applicable to class frames only, but the selection consists of a mixture of class and instance frames, only the class frames are used. If a selected frame cannot be used, a warning message is printed in the text pane.

If no frame is selected when an operation that requires one is invoked, you are prompted to enter a frame. If just one frame is required, you are prompted directly. A frame can be entered either by clicking on the corresponding node in the graph pane, or by typing in its name in the text pane. If you change your mind, and no longer want to perform the operation, typing `Control-Z` or right clicking in the background aborts the command. If the operation can accept more than one frame, you may be asked to input a sequence of frames. You can do this either by left clicking on several frames in sequence, or entering a comma-separated list of frame names via keyboard. In some cases, additional selection options are available (such as **All** or **None** or **Compute Automatically**), in which case, a menu of options pops up. You may choose to enter only a single frame, in which case the frame is entered as before. If you choose to enter multiple frames, you are put into a mode in which every node you left click on is added to or removed from the selection. You cannot type in frame names in this mode. Left clicking on the background ends the selection process. If the command can accept no frames (as in the case of creating a class with no parents), this is also provided as an option in the menu. The **No Select** option in the menu aborts the command.

When selecting an edge, if two nodes are preselected and there exists an edge between them, that edge is used. If one node is preselected, it is assumed to be the child node (unless it is a root, in which case it is assumed to be the parent). If there is only one incoming edge, that edge is selected automatically. If there is more than one incoming edge, you are asked to click on the desired parent. If no nodes are preselected, you are asked to select the child first, and then the parent if necessary.

Completion is available in most cases when you type in frame and slot names: if you enter a partial name and then type `Return`, a menu of completions pops up.

8.3.3 The Kill Ring

A cut-and-paste/kill-ring facility is available for use in most input contexts, to reduce the need to type in frame names and so forth. Any time the **Copy** to **Kill Ring** command is invoked, the currently selected nodes are copied to a kill ring. A node is also copied to the kill ring when it is individually deleted. When prompted to enter

a frame or a value. `Control-Meta-Y` yanks the last object from the kill ring. `Meta-Y` invoked immediately after a `Control-Meta-Y` or another `Meta-Y` rotates the kill ring. This facility attempts to yank values in the form expected by the input context. For example, if you are being prompted to enter a frame, the yanked value is coerced to a frame name where possible. If you are prompted to enter a string, the value is converted to a string and so forth.

Because of limitations beyond our control, the kill ring cannot handle multiline strings, and also is not usable from within a Motif text-editing window. Motif text-editing windows are used from within the frame-editing viewer to edit multiline values, which the kill ring cannot handle, anyway, becuase of the first limitation. You can add a variety of emacs-like editing commands within Motif text-editing windows by adding the appropriate entries to your `.Xdefaults` file.

8.3.4 Creating Classes and Slots

If you are creating a new KB from scratch, the basic pattern to follow is to first create the classes, then create the slots, and finally, create the instances and store slot values in the instances. In fact, you can interleave these different tasks. The more precise constraints are that an instance cannot be created until at least one of its parent classes exists, and a slot cannot be created until the most general class that will contain the slot has been created.

Once you have created classes and slots, the degree to which you can modify them within the GKB Editor may be constrained by your FRS.

Creating slots is a bit tricky, so we discuss the issue in more detail. Most FRSs store not only the values of slots within a given frame, but they store meta information about each slot that describes general properties of how the slot will be used in the KB. These slot properties are often called *facets*; facets are in a sense "slots on slots" because facets store a set of named values that are associated with a slot. For example, a facet called `:value-type` defines the types of values that a given slot can take, and a facet called `:domain` defines those KB classes within which a slot can appear. The facet names listed here may not actually be used within your FRS, but because each FRS uses different facet names, the GKB Editor internally translates from the names shown here to the names used by your FRS.

All editing of slots is performed by invoking the Frame Editor on the class frame that will contain, or that already contains, the slot in question. For example, to add a slot called `Age` to the class `Person`, first invoke the Frame Editor on `Person`. Then click on **Create** from the **Slot Menu** to add the `Age` slot to this class. Similarly, the **Destroy** operation removes `Age` from `Person`, and the **Rename** operation renames the slot.

More subtle modifications to a slot definition can be achieved by changing its facets. Imagine that we want to state that the `Age` slot can take only one value,

which must be an integer. To do so, we would give the `:cardinality-max` facet the value 1, and we would give the `:value-type` facet the value `:integer`.

These facet values can be assigned in two ways. To establish default, KB-wide values of the facets, use the **Edit Slotunit** operation from the **Slot Menu**. A *slotunit* is a special frame whose slots encode KB-wide default values for slot facets. For example, if we edit the slotunit for `Age`, and give its `:value-type` slot the value `:integer`, we are effectively setting the `:value-type` facet for the `Age` slot.

The other approach is to edit the facets of `Age` directly within a class frame. Use this approach when either your FRS does not support slotunits, or when you want to specialize facet values. For example, imagine that our FRS also supported a facet called `:numeric-max`, which defined the maximum value that a slot value is allowed to have. In the `Age` slotunit, we could set the value of `numeric-max` to 150 – a reasonable value for the class `Person`. Imagine that we now define a new class, `Infant`, for which the `:numeric-max` facet of `Age` should be further constrained to have the value 1. To do so, first invoke the Frame Editor on class `Infant`. Then use the **Preferences Menu** to enable the display of facets. If your FRS supports the **:numeric-max** facet, you can then click on it and use the **ReplaceValue** command to change its value in that class, without affecting the **:numeric-max** of `Age` in `Person`.

8.3.5 Invoking Viewers Programmatically

You may want to incorporate part or all of the GKB-Editor into another Lisp application by invoking one or more of the viewers programmatically from your other Lisp application. The Lisp commands to accomplish this are described here.

`gkb-editor:view-hierarchy (&key start-browse? roots width height)` Invoke the hierarchy viewer. Since this is the main GKB-Editor startup window, it also provides access to the other viewers from its command menus. If `start-browse?` is non-`nil`, and there exists a current open KB, then the GKB-Editor immediately begins browsing the class hierarchy. If `roots` (a list of frame names) are supplied, then they are used as the roots of the hierarchy browse (assuming `start-browse?` is non-`nil`): otherwise the roots are computed using your current preferences, as usual. If `start-browse?` is `nil` and a previous hierarchy browse exists, it is resumed; otherwise the viewer appears showing the GKB-Editor copyright graph. If both `width` and `height` are supplied, then they are used for the dimensions of the viewer window; otherwise, the dimensions are automatically calculated based on your screen size.

`gkb-editor:view-relationships (&key roots width height)` Invoke the frame relationships viewer. If `roots` (a list of frames) is specified, the specified frames are used as roots of the relationships browse. Otherwise, if a previous relationships browse exists, then that browse is resumed. If no roots are specified and there is no browse to resume, you are prompted to enter the roots. If both `width` and `height`

are supplied, then they are used for the dimensions of the viewer window; otherwise the dimensions are automatically calculated based on your screen size.

`gkb-editor:view-frame (&key frame slot (width 800) (height 500))`
Invoke the frame-editing viewer. If `frame` (a frame name) and `slot` (a slot name) are both specified, then only the specified slot and its values, facets, and so forth are displayed. If `frame` is specified, but not `slot`, then the entire frame is displayed. This is what is normally seen when you invoke the frame-editor from one of the other viewers. If `slot` is specified, but not `frame`, and a slotunit for the specified slot exists, the slotunit information appears in the display. If neither is specified, this function returns immediately with no effect. `width` and `height` are used for the dimensions of the viewer window.

8.3.6 Miscellaneous

Most commonly used commands have **keyboard accelerators** defined for them. These commands are therefore accessible either by using the mouse to select the appropriate command from a menu or by typing some key combination (usually some letter while you hold down either the Control key, the Meta key, or the Control and Shift keys) on the keyboard. Commands with keyboard accelerators have the corresponding key sequence listed next to the command name in the menu, and also in this documentation. As keyboards differ, you can experiment with your keyboard to determine which actual key corresponds to the Meta key.

In any viewer, you can use `Control-v` to scroll the viewport downward and `Meta-v` to scroll it upward.

8.4 The Class Hierarchy Viewer

The class hierarchy viewer is the viewer that you are in by default when you first start up the GKB-Editor. It allows you to incrementally browse the class-instance taxonomy. The class hierarchy is drawn as a tree-structured graph in which each frame is a node in the graph, and directed edges are drawn from classes to subclasses and instances. If a frame has multiple superclasses, an edge is drawn from each superclass to the frame. Classes are distinguished visually from instances. By default, browsing starts at the root of the class hierarchy. This can be changed, however. A sample hierarchy browse is shown in Figure 8.1. Classes are in red, and instances are in blue. Notice that some class labels are in bold type, and there are some additional nodes in black labeled `N more`. These are described in Section 8.4.3.

From the class hierarchy viewer, you can create, delete, and rename classes and instances, as well as change the position of any frame within the class-instance hier-

archy. You can also perform operations on the KB, change various preferences, and start up one of the other two viewers.

8.4.1 Opening a KB

Initially, when you start up the GKB-Editor, a KB is probably not open yet (even if you have loaded a KB by other means before starting the GKB-Editor, the KB is usually not visible to the GKB-Editor unless you officially open it using the GKB-Editor or with GFP commands). If no KB is open, most of the menu commands are disabled, and the status line contains the message `Create or Open a KB`. To create or open a KB, select **New** or **Open** from the **Knowledge Base Menu**. If creating a new KB, you are prompted for a name, package, and filename. If opening an existing KB for the first time, you are prompted for a filename. After determining that the file exists, and trying to determine the KB name and package from the file contents, the GKB-Editor asks you to verify the KB name and package before it opens the KB.

The GKB-Editor stores a record of KBs that you have opened in the past, so in subsequent sessions when you invoke the **Open** command, as a shortcut, you are offered a menu of KBs that you have previously worked with (if they still exist). The menu also has an option (`Other...`) to select a KB not on the list, in which case you must go through the set of dialogs described above.

If your FRS permits you to have multiple KBs open at once, the **Select** command enables you to switch between open KBs. Other operations that are accessible from the **Knowledge Base Menu** enable you to save, revert, close, or destroy a KB. A few KB analysis tools are available.

8.4.2 Knowledge Base Commands

- **New** (`Meta-o`)
  Create, open, and select a new KB. You are prompted in a dialog for a name, package, and file in which to save the new KB. If multiple supported FRSs are present in memory, you must also select which FRS to use. The new KB has no frames in it. Consequently, the only frame operation permitted at this point is class frame creation.

- **Open** (`Ctrl-o`)
  Open and select a preexisting KB. A menu of known KBs pops up. You can choose one of these KBs or be prompted for a filename for the KB to open. The KB is then loaded into memory. It is an error for the KB not to exist in the specified file.

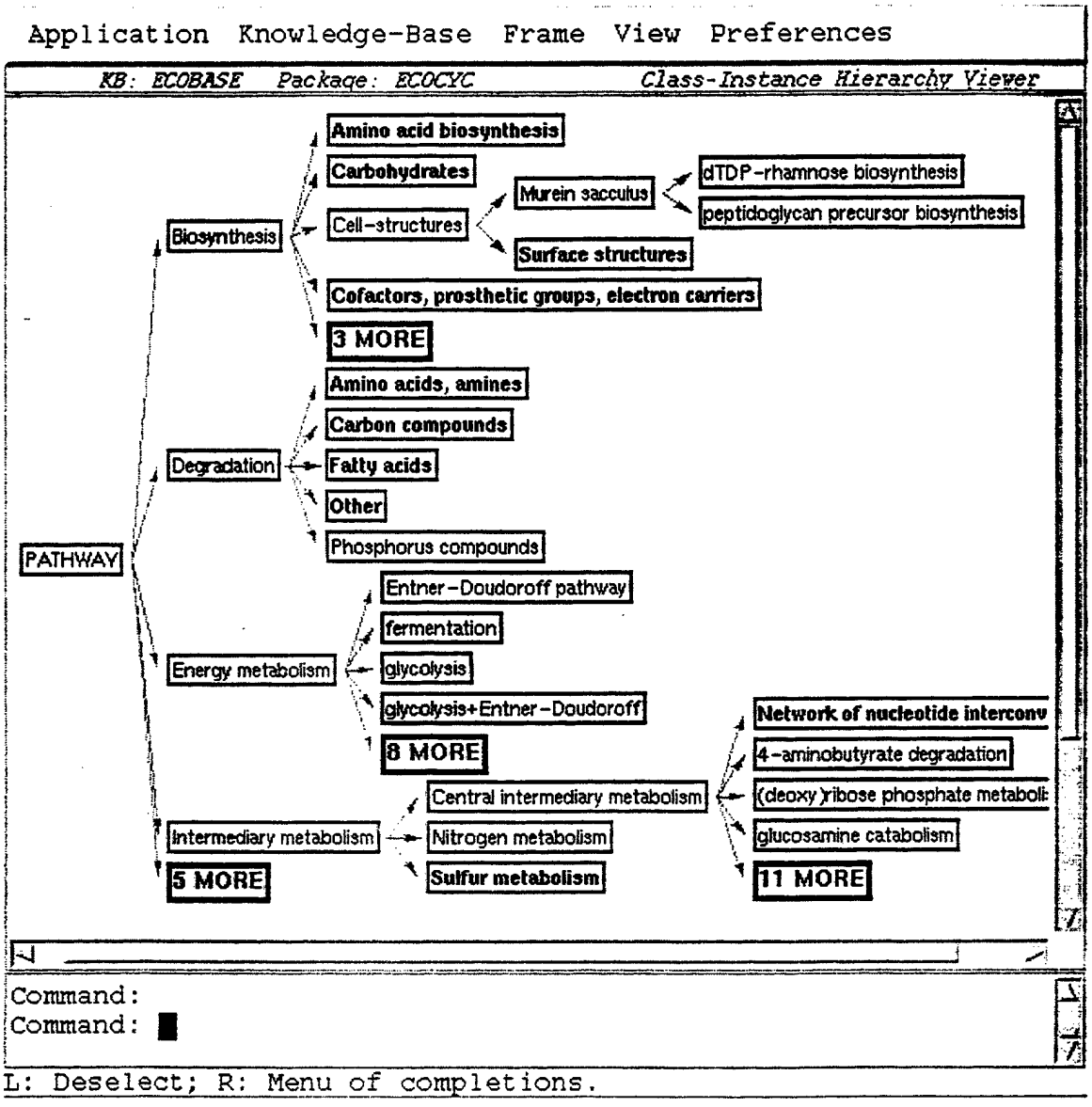


Figure 8.1: A sample class/instance hierarchy view.

- **Select** (`Ctrl-O`)
  A menu of currently open KBs pops up. The selected KB becomes the current KB to browse and edit.

- **Save** (`Ctrl-s`)
  Save the current KB to the file specified when the KB was created, opened, or last saved.

- **Save As** (`Ctrl-S`)
  Save the current KB to a new file. You are prompted for a filename to save to.

- **Revert to Saved**
  Reload the KB from secondary storage. Any changes made since the last save are lost.

- **Close**
  Close the current KB and flush it from memory. If the KB has been modified, you are prompted about whether or not to save it first.

- **Destroy**
  Close the current KB and delete the associated file on secondary storage.

- **Analyze**
  This command collects statistics on the current KB and displays them in a popup window. The types of statistics generated include number of classes and instances, depth of the class hierarchy, percentage of primitive classes, average number of slots per frame, and average number of values per slot.

- **Analyze Subtree**
  This command collects statistics on a subtree of the KB class hierarchy and displays them in a popup window. If you have selected a class frame, it is the root of the analyzed subtree. If no frame has been selected, you are prompted for the root.

8.4.3 Incremental Browsing

To begin browsing initially, select the **Browse Class Hierarchy** command from the **View Menu**. You can also begin browsing by selecting the **Browse from New Root(s)** command: this command requires that you enter the name of a frame to serve as the root. Subsequent browses invoked using the **Browse Class Hierarchy** command then use this same root. The GKB-Editor has a sophisticated incremental browse facility that allows you to reduce screen clutter and screen redisplay time by drawing only the parts of the KB that you want to see. Thus, initially, you see only the top few levels of the class hierarchy, and classes with many children show only

a truncated list of children. The degree to which the graph is initially expanded is determined by a depth limit and a breadth limit, which you can change using the **Incremental Expansion** dialog (you can also change the way the root nodes are chosen with this command). If a node has children that are not displayed because of the depth limit, that node, called a *depth cutoff node*, is displayed in boldface. If a node has more children than the breadth limit, then only the number of children up to the breadth limit is displayed. An additional node, a *breadth cutoff node*, is drawn with the boldface label `N more`, where N is the number of remaining children. Some depth and breadth cutoff nodes are shown in Figure 8.1.

It is possible to also specify whether or not the depth limit is *hard*. If a hard limit is used, then expanding one branch of the tree causes expanded branches in other sections of the graph to be compacted. This allows you to focus in on only the current region of interest.

To expand a depth or breadth cutoff node, middle click on the node. Expanding a depth cutoff node causes its children to be drawn. Expanding a breadth cutoff node, or a node that has a breadth cutoff node for a child, causes the breadth limit for that node to double, and the additional children are drawn up to the new breadth limit. Attempting to expand a node that is neither a depth nor breadth cutoff has no effect. Middle clicking on a node while holding the Shift key down compacts the node, erasing all its children and turning that node into a depth cutoff node. Right clicking on a node brings up a menu of operations. The possible operations (not all operations may be applicable to every node) include

- Select children to expand
- Expand all children
- Expand all descendants
- Expand parents
- Erase node and its descendants
- Change browsing parameters

You can expand the browse all the way to some frame, without knowing exactly which sequence of classes needs to be expanded in order to find the frame. To do this, invoke the **Find** command from the **Frame Menu**. You are prompted (with completion) for the name of a frame to find. If the frame is present in the KB, the current browse is expanded to make it visible, and the frame is highlighted. Sometimes, no ancestor of the specified frame is currently visible in the display, in which case the found frame is shown as a root frame with no parent classes. In

this situation, you must use the *Expand parents* option in the right click menu to determine the frame's place in the class hierarchy.

You can also save a browse to a file, so that in subsequent sessions you can begin browsing exactly where you left off.

8.4.4 View Commands

- **Browse Class Hierarchy** (`Ctrl-b`)
  Begin a new browse of the class-instance hierarchy. Any previous expansions are lost. Changes to browse parameters (i.e., through the preference menus) that do not take effect immediately will take effect after this command.

- **Browse from New Root(s)** (`Meta-b`)
  Begin a new browse of the class-instance hierarchy. You are prompted for one or more frames to serve as roots of the new hierarchy. This command also changes your preferences such that subsequent browses invoked using the **Browse Class Hierarchy** command begin with the selected nodes.

- **Redraw**
  Refresh the window.

- **Save Browse**
  You are prompted for a filename. The current browse state (i.e. which nodes are visible) is saved to the designated file.

- **Load Browse**
  You are prompted for a filename containing a previously saved browse. The browse is reloaded and displayed. If the state of the KB has changed since the browse was originally saved, the loaded browse may have changed in appearance accordingly.

8.4.5 Editing Operations

The Class Hierarchy Viewer enables you to create new frames and to rename, delete, and change class-subclass and class-instance links for existing frames. If an editing operation acts on a particular frame, you are required to select the frame either before or after selecting the command. See Section 8.3.2 for details. All the editing commands can be found under the **Frame Menu**. To perform additional editing operations on an individual frame, such as altering its slot values, you must invoke the Frame Editing Viewer, which you can also do from this menu.

8.4.6 Frame Commands

The following commands all operate on one or more frames. The first few commands are not actually editing operations, but they may facilitate selecting frames on which to perform editing operations.

- **Select All**
  Make all currently displayed frames part of the current selection.
- **Select Descendants**
  Add all currently displayed descendants of currently selected nodes to the current selection.
- **Select Ancestors**
  Add all currently displayed ancestors of currently selected nodes to the current selection. This operation is also useful for highlighting the path of a particular frame back to the root.
- **Find** (`Ctrl-f`)
  You are asked to type in a frame name. If the frame is already visible in the graph, it is highlighted. If the frame is not yet visible, then the browse is expanded until the designated frame becomes visible.
- **List Frame Contents** (`Ctrl-l`)
  A list of direct superclasses, subclasses, and instances, and all slot values is displayed in a new window. Click on `Dismiss` to close the window. This may be faster, or may result in a more compact display, than invoking the Frame-Editing Viewer in cases in which you want to only examine rather than modify the contents of a frame.
- **Edit** (`Ctrl-e`)
  Invoke a Frame-Editing Viewer to edit the selected frame.
- **Edit Instance of Class** (`Ctrl-E`)
  For situations in which you are displaying classes only, or do not want to expand a class node, this command presents a menu of instances of the currently selected class frame. The chosen instance may be edited using the Frame-Editing Viewer.
- **Edit Loom Definition**
  This command is provided for users of Loom because the GFP does not provide support for all the possible types of constraints and restrictions available in Loom. A text editing window pops up, containing the Loom definition of the selected frame. It can be edited using basic Emacs commands. Middle clicking in the editing window closes the window, and causes the selected frame to be redefined with the new definition. In general, changes made in this fashion do not immediately update the graphical display.

- **Create Class** (`Ctrl-n`)
  Create a new class frame. The selected class frame(s) are used as the superclass(es) of the new class. You are prompted for a name for the new frame in a small pop-up window.

- **Create Instance** (`Ctrl-N`)
  Create a new instance frame. The parent(s) of the new instance is the selected class frame(s). You are prompted for a new name in a small pop-up window.

- **Create Duplicate** (`Ctrl-d`)
  Create a new frame as a copy of the selected frame. If the selected frame is a class, the new frame is a class; if the selected frame is an instance, the new frame is an instance. The new frame has the same parents and slot values as the original frame. You are prompted for a name for the new frame in a small pop-up window.

- **Rename** (`Ctrl-m`)
  Rename the selected frame. You are prompted for a new name in a small pop-up window.

- **Destroy** (`Ctrl-k`)
  Destroy the selected frame(s). If a class frame is destroyed, any subclasses or instances of that class become children of the original frame's parent(s).

- **Destroy Subtree** (`Ctrl-K`)
  Destroy the selected frame(s). If a class frame is destroyed, any subclasses or instances of that class are also destroyed. When this occurs, you are first prompted to make sure that is what was intended.

- **Add Parent Link**
  You are asked to select a new class frame to serve as a parent of the selected frame, and the new link is added.

- **Change Parent Link**
  You areasked to select a link to change by first selecting the child frame and then the old parent (if necessary), and then to select a class frame to serve as the new parent. The old link is deleted, and a new link is created from the new parent to the old child.

- **Delete Link**
  You are asked to select a link to delete by first selecting the child frame, and then the parent (if necessary). In the hierarchy viewer, the parent - child link is deleted. If the old chiid frame has no remaining parents, it becomes a root.

8.4.7 Customizing the Display

The GKB-Editor is highly customizable, allowing users to specify via a set of preference menus what should be displayed in the various viewers and how various objects should be drawn.

In the Hierarchy and Relationships Viewers, you have a great deal of control over how individual nodes are displayed. We provide a style menu that allows you to specify icon and label colors, icon shape, and label font, face, and size. You can define a style for all frames satisfying some predicate. Any number of these styles and predicates can be defined, and frames that satisfy more than one such predicate 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, andprimitive classes, for testing frames for a particular user-specified slot value, or identifying all children of a particular class. Future versions of the GKB-Editor may support a greater variety of predefined predicates, or may permit arbitrary, user-defined predicates. Only two styles are defined when the GKB-Editor is first installed, for class and instance frames, respectively.

Preferences are made persistent by saving them in a *profile*. Your profile is saved in the file `.gkb-prefs` in your home directory. There are two types of profiles: the *User Profile*, which contains preferences that should be used for all KBs that you open, and the *KB Profile*, which contains preferences that apply only to a particular KB that you open. One can also imagine a *Global KB Profile* that applies to a particular KB for all users, but this has not been implemented. You can have only one User Profile, but many KB Profiles, one for each KB. Preferences specified in the KB Profile override those in the User Profile when that KB is open.

When saving preferences, in general you are given a choice of whether a particular changed preference should be a part of the User Profile, the KB Profile, both, or neither. However, some preferences, such as those that refer to a particular frame in a KB, can be saved only as part of a KB Profile. Upon invoking the **Save Preferences** command, you are presented with a dialog that lists the various options.

8.4.8 Preference Commands

- **Incremental Expansion**
  Presents a dialog through which you can alter browsing parameters (see Section 8.4.3). You can designate new depth and breadth limits, and whether or not a hard depth limit should be used. When the hard depth limit is enforced and exceeded along a given branch of the class hierarchy, all other branches of the hierarchy are collasped so the particular branch can be browsed more intensely. You can choose whether the browsing root on a new browse should be computed automatically, should be set to some default, or should be taken

from your input. If you want to set a default, then upon exiting the dialog, you are asked to input a frame or series of frames to serve as the default root(s). You can also decide whether the browse should show class frames only, or both classes and instances. Some of these changes do not take effect until a new browse is initiated.

- **Node Styles**

  - **Select Node Label Slots**
    For easier viewing, you may want to include certain slot values as part of the node labels. This command presents a menu of all slots in the KB and allows you to choose which slots should be included in the node labels.

  - **Redefining a Node Style**
    The list of currently defined node styles is presented. You can change the style definition for a particular style by clicking on the "Redefine Style" command button after selecting a style from the provided list. A dialog appears with the current definition, which you can change. The style dialog lists various options for icon shape and color, and label size, color, and font. If a particular option is left unspecified, or Default, then the default value for that node is used. This allows nodes to be displayed using multiple shapes. For example, if grouping A specifies only the icon shape, and grouping B specifies only the label color, then a node that is a member of both groupings is drawn with icon shape A and label color B. However, if grouping B instead specified the icon shape, the node could be drawn only as belonging to either grouping A or B, and not both. For this reason, we encourage you specify different options when defining styles for orthogonal groupings, and to use the Default value for all other options. You can use the "Apply Style" button to try out different style combinations before exiting the dialog.

  - **Removing a Node Style**
    Style information and grouping definition for the selected style can be removed by using the "Delete Style" command button. This button is enabled when a selected item can be deleted from the list of existing styles. Upon deletion of the style, its definition name is removed from the list and may be added to the list of possible styles you can create in the future.

  - **Defining a New Node Style**
    The list of possible but yet undefined node styles is presented. You can assign a node style to a new grouping of frames by applying the "Define Style" command button to a selected item on this list. You are offered a dialog of options (described above) with which to define the new style. You can define styles (if not already defined) for class and instance frames, primitive and nonprimitive class frames, all children of a particular class,

and frames with a particular value for a particular slot. If defining a style for all children of a particular class, youare asked to type in the name of the class. If defining a style for frames with a particular slot value, you are asked to type in first the slot name, and then the desired slot value.

- **Miscellaneous**
  The miscellaneous dialog asks you whether or not the frame names should be case sensitive. Additional options may be added to this dialog in a later release of the GKB-Editor.

- **Save Preferences**
  You are presented with a dialog that allows you to save any changed preferences to either your User or KB Profile. The next time you use the GKB-Editor to access the same KB, these preferences automatically take effect. If you change your preferences and forget to save them, you are asked before you exit the session whether or not they should be saved.

- **Revert to Saved**
  Revert to the preferences as of the last time they were saved. Any preference changes made since then are undone.

- **Revert to Default**
  Revert to the GKB-Editor default preferences. Any saved preferences are undone in the current session, but, unless you save preferences again, are available for subsequent sessions.

8.4.9 Application Commands

Miscellaneous commands found in the **Application Menu** relate either to the application as a whole (i.e., **Quit**, **Print** or **Help**), or to invoking the other types of viewer. Keep in mind that when opening additional viewer windows, only the most recently created window is active and accepting commands. For example, if you open a relationships viewer in the middle of browsing the class hierarchy, you must close the relationships viewer before you can continue browsing the class hierarchy. We hope to overcome this limitation in a future release.

- **Invoke Frame Relationships Viewer** (`Ctrl-r`)
  Open up a relationships viewer window and begin a new relationships browse. The selected frame(s) is the root of the relationship tree. If you have not previously selected which slots to expand, all slots are expanded.

- **Resume Frame Relationships Browse** (`Ctrl-R`)
  Open up a relationships viewer window, and continue the relationships browse

where it left off the last time a relationships viewer window was open. This command enables you to switch back to a browse of frame relationships after browsing the class hierarchy. If no current relationships browse exists, this command has no effect.

- **Invoke Frame Editor** (`Ctrl-e`)
  Invoke a Frame-Editing Viewer to edit the selected frame. This command is identical to the **Edit** command in the **Frame Menu**.

- **Print Display** (`Ctrl-p`)
  Print the graph window to a postscript file.

- **Print Report**
  The Print Report command is available only in the taxonomy viewer. The command provides a textual reference of parts or all of any knowledge base available to you. When this command is invoked, you are presented with a dialog allowing you to specify the name of the file the report should be written to (the default is "kbName.txt"). You can then choose from a list how much of the knowledge base should be printed to the file. You can choose to have a textual reference of the entire KB or just selected frames. Other options are presented in the list. You can also specify which order the information should be written in, either alphabetical, depth first, or breadth first. This order may or may not apply to all the options given in the list of parts of the KB to print. For example, if you choose to have only the instances printed, it does not make sense for them to come in depth-first order. They are automatically printed in alphabetical order. You are also asked which slots and facets to include in the frames being printed to the file.

- **Copy to Kill Ring** (`Ctrl-c`)
  Copy the current selection to the kill ring.

- **Help**
  Display this user manual in your Netscape browser.

- **Quit** (`Ctrl-Q`)
  Exit the GKB-Editor, returning control to the Lisp Listener.

- **Kill**
  Exit the GKB-Editor, and kill the Lisp process (i.e., the Unix process) in which it was running.

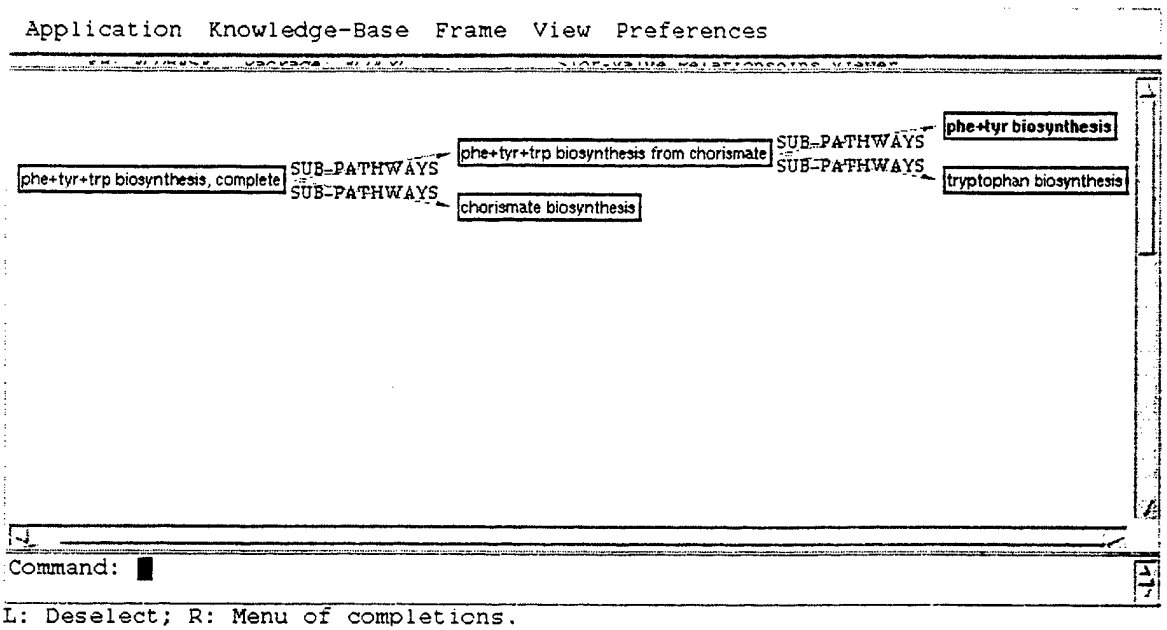


Figure 8.2: A sample frame relationship view.

8.5 The Frame Relationship Viewer

Whereas the Class Hierarchy Viewer enables you to browse a KB following class-subclass and class-instance relationships, the Frame Relationships Viewer provides a means of browsing the KB by following slot-value relationships. For example, in a KB of automobiles, you may have defined a `Part` slot, such that values of the `Part` slot for a particular car frame include a `Wheel` frame and an `Engine` frame. The *Part* slot values for the `Wheel` frame may include a `Tire` frame and a `Bearing` frame, and so on. One can imagine constructing a graph showing the entire *Part* hierarchy starting at some car. This is exactly what the Frame Relationships Viewer does. You can specify which slot-value relationship or relationships to follow (or just follow all of them), and one or more instances to start from, and a tree is drawn with the specified instances as roots. A frame B is drawn as a child of frame A if B is a slot value of one of the designated slots in A. Each edge is labeled with the appropriate slot name.

Figure 8.2 is an example of a relationships slot-value browse in a biochemistry KB, showing how a super-pathway is composed of its sub-pathways. The edges are labeled with the slot name, to distinguish slots so that you can follow more than one slot at a time.

An alternative way of browsing frame relationships is to look purely at classes, that is, to determine which classes a value of a particular slot must be a member of. For example, if a slot in class A must be filled by an instance of class B, then class B is drawn as a child of class A. This type of browsing is used if all the selected root

frames are classes. We call this type of browse a **slot-type** browse, to distinguish it from the **slot-value** browse described earlier.

Various editing operations are available from this viewer. In addition to creating, deleting and renaming frames, slot-value relationships can be inserted and deleted by adding or removing the appropriate edges. This facility provides an intuitive mechanism for building up complex interrelationships between frames.

To invoke the frame relationships viewer, select one or more frames and the command **Invoke Frame Relationships Viewer** from the **Application Menu**. Note that because you must initially select one or more frames to serve as roots of the relationships browse, it may help to browse the class hierarchy first, to bring the desired frames into the display where they can be clicked on. If you do not browse the class hierarchy first, then after selecting the command, you are prompted for a frame: just type in the name of the desired frame or frames in the text pane. A new application window containing the relationships view pops up and becomes active. You must close this window before you can resume browsing the class hierarchy or performing any other commands in the class hierarchy viewer.

By default, all slots that contain frames as values are selected for browsing, but you can change this if you are interested in only one or a few slots.

If any of the selected roots are instance frames, then the browse will be a slot-value browse. If all selections are class frames, the browse will be a slot-type browse.

The same incremental browse facility is available for the relationships viewer as for the Class Hierarchy viewer. See Section 8.4.3 for details. The command menus also contain many of the same commands as in the Class Hierarchy Viewer command menus.

8.5.1 Application

The commands **Print Display**, **Copy to Kill Ring**, and **Help** are as in the class hierarchy viewer. The one additional command, **Close** (`Ctrl-q`), closes the frame relationships viewer window and returns control to the hierarchy viewer window.

8.5.2 Knowledge Base Commands

When this viewer is invoked, a KB should already be open and selected. Thus, only a subset of the commands from the class hierarchy **Knowledge Base Menu** is available in this viewer: **Save**, **Save As**, and **Revert to Saved** allow you to save the KB, to save the KB to a specified file, and to revert to the last saved version of the KB, respectively.

8.5.3 Frame Commands

The following commands all operate on one or more frames. There is some duplication here: some of the editing commands can also be performed from the hierarchy viewer, whereas others can be performed from the frame-editing viewer.

- **Find** (`Ctrl-f`)
  You are asked to type in a frame name. If the frame is already visible in the graph, it is highlighted. If the frame is not yet visible, then it is made visible in the graph in the following way: if the frame is a direct slot-value child of some frame currently visible in the graph, then the frame is shown as a child of the currently visible frame: otherwise, the frame is added as a new root in the graph. In other words, no attempt is made to expand the graph more than one level deeper than its current depth in order to locate the frame.

The following eight commands are identical (except possibly in their effects on the display) to the analogous commands from the hierarchy viewer.

- **List Frame Contents** (`Ctrl-l`)
- **Edit** (`Ctrl-e`)
- **Edit Loom Definition**
- **Create Class** (`Ctrl-n`)
  In this and the next two commands, the new frame becomes a root in the current browse.
- **Create Instance** (`Ctrl-N`)
- **Create Duplicate** (`Ctrl-d`)
- **Rename** (`Ctrl-m`)
- **Destroy** (`Ctrl-k`)
- **Add Relationship** (`Ctrl-a`)
  You are asked first to select the slot to add to (if there is more than one), and then to select the frame to link to. If the frame to link to is already present in the display, the link is added. If the frame to link to is not yet present, it is added as a child of the first frame. If the frame to link to does not yet exist, you are given the option of creating it. The parent class of the new frame is derived from the range of the selected slot.

- **Delete Relationship**
  You are asked to select a link to delete by first selecting the child frame, and then the parent (if necessary). The slot-value link is then removed. If the old child frame has no remaining parents and no visible children, it is removed from the display. If it has no remaining parents, but has visible children, it becomes a new root in the display.

8.5.4 View Commands

These commands pertain to setting up a relationships browse, and which frames are pictured on the display. They are generally analogous or equivalent to the **View** commands in the hierarchy viewer.

- **Restart Browse** (`Ctrl-b`)
  Begin a new browse of the frame relationships hierarchy, using the same root frames as before. Any previous expansions are lost. Changes to browse parameters (i.e., through the preference menus) that do not take effect immediately do take effect after this command.

- **Browse from New Root(s)** (`Meta-b`)
  Begin a new browse of the frame relationships hierarchy. You are prompted for one or more frames to serve as roots of the new hierarchy. This command also changes your (nonpersistent) preferences such that subsequent browses invoked using the **Restart Browse** command begin with the selected nodes.

- **Change Slots**
  This command pops up a menu of all slots available in the current KB. You should select the slots that you want the relationships browse to follow. A new browse is begun from the original root frames, following the selected slots. Any previous expansions of the graph are lost. Future relationships browses will also use this set of slots, until the set of slots is changed again. However, the choice of slots is not saved persistently and is not in effect in subsequent GKB-Editor sessions. To persistently change slots, use the Incremental Expansion dialog in the **Preferences Menu**.

The following three commands are identical to the analogous commands from the hierarchy viewer.

- **Redraw**
- **Save Browse**
- **Load Browse**

8.5.5 Preference Commands

- **Incremental Expansion**
  This command presents a dialog through which you can alter browsing parameters (see Section 8.4.3). You can designate new depth and breadth limits, and whether or not a hard depth limit should be used. These parameters are the same as for the taxonomy viewer. Selecting slots to expand applies only in the relationships viewer. You are presented with a menu of all slots in the current KB that can take other frames as slot-fillers. All selected slots are expanded in a relationships browse. This information is displayed in place of the browsing parameters you are presented with in the taxonomy viewer.

The following commands, which are found in the preferences dialog, are identical to those found in the taxonomy viewer.

- **Node Styles**
- **Miscellaneous**
- **Save Preferences**
- **Revert to**
- **Revert to Default**

8.6 The Frame-Editing Viewer

From either the hierarchy viewer or the relationships viewer (or another frame-editing viewer), it is possible to select a frame for editing. Invoking the **Edit** command from the **Frame Menu** of either of the other two viewers causes a Frame-Editing window to pop up. In this viewer, each slot name forms the root of a small tree, with its children being the individual slot values and any slot facets (defaults or constraints) that you want to display. If the FRS is one that supports annotations on values, annotations are drawn as children of the appropriate values. Inherited items are distinguished visually from local items, and cannot be edited (but they can be overridden where appropriate).

The kinds of editing commands available in this viewer include adding, deleting, and editing slots and their values, facets, and annotations. From the frame-editing viewer, you can also recursively bring up new frame-editing viewers for frames and slots in the current display.

Figure 8.3. is an example of a frame editing window. The name of the frame being edited is shown in bold just below the command menus. Slots are listed at the

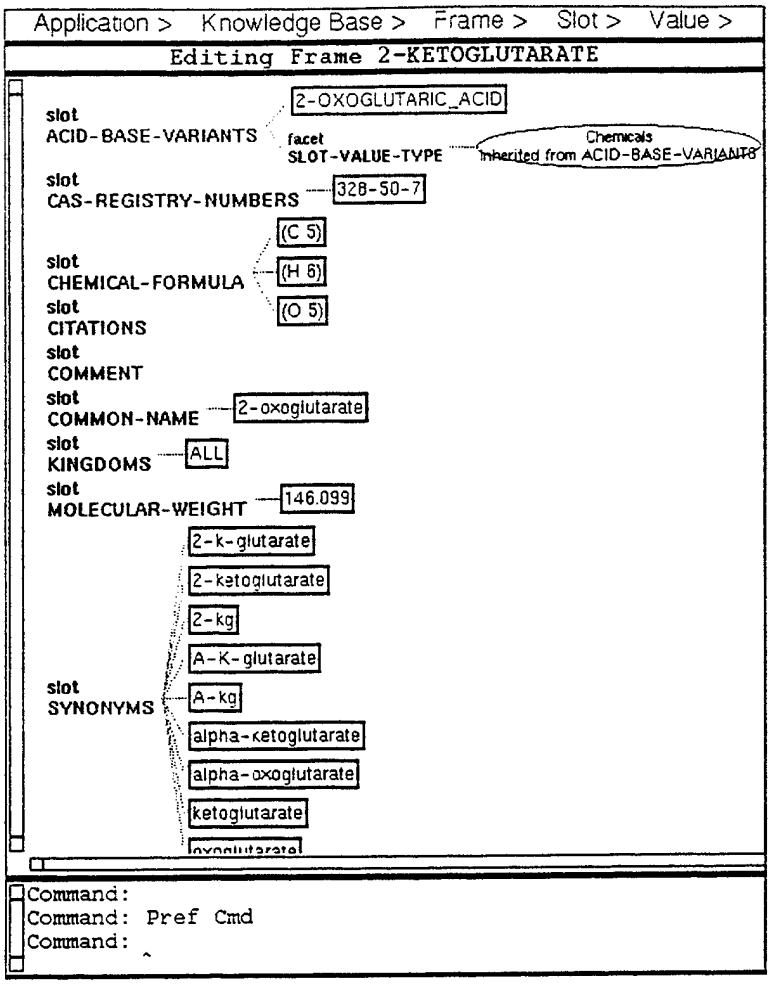


Figure 8.3: A sample frame view.

left of the window, with links to their values and facets. Slot and facet names are tagged by the word *slot* or *facet*. Slot values appear in a rectangular box. Facet values appear inside an ellipse. Local values are shown inside a bold icon, whereas inherited values are not. In this example, showing the frame `2-KETOGLUTARATE` (a biochemical compound), the slot `ACID-BASE-VARIANTS` has one value, `2-OXOGLUTARIC_ACID`, which is a local value. One facet is shown, `SLOT-VALUE-TYPE`, which constrains values of the slot to be of a certain type. In this case, the facet value is `Chemicals`, meaning that all values for this slot must be chemicals. The facet value is an inherited constraint. Since in this example we have chosen to show the source of inherited values, we can see that the constraint derives from the slot's definition (the slotunit). If a frame system supports annotations on values, these are shown as children of their associated values. No annotations are shown in this example.

Like the relationships viewer, the frame-editing viewer pops up in a modal fashion: while it is active, no previous viewers accept commands. To resume the previous

browsing or editing activity, you must first close the current frame-editing viewer.

A keyboard shortcut has been provided for selecting a slot. Pressing the Space key selects the first slot. Pressing the Space key again changes the selection to the next slot, and so on. The Backspace key changes the selection to the previous slot. This facility is useful when you want to iterate over the available slots, adding a value or values for each one, without using the mouse.

8.6.1 Application Commands

The commands **Print Display**, **Copy to Kill Ring**, and **Help** are as in the class hierarchy and frame relationships viewers. The command **Close** (`Ctrl-q`) closes the frame-editing viewer window and returns control to the viewer window from which it was invoked.

8.6.2 Knowledge Base Commands

When this viewer is invoked, a KB should already be open and selected. Thus, only a subset of the commands from the class hierarchy **Knowledge Base Menu** is available in this viewer: **Save**, **Save As**, and **Revert to Saved** allow you to save the KB, to save the KB to a specified file, and to revert to the last saved version of the KB, respectively.

8.6.3 Frame Commands

While most of the editing commands accessible in the frame-editing viewer apply to parts of frames (e.g., slots, values), these commands act on whole frames. The first three commands in this menu manipulate the frame currently being edited. The last two commands allow a different frame to be edited by invoking a new frame-editing viewer.

- **Rename**
  Rename the selected frame. You are prompted for a new name in a small pop-up window.

- **Destroy**
  The current frame is deleted from the KB. This closes the current frame editing window.

- **Edit Loom Definition** (for Loom only)
  Edit the Loom definition of the frame in a new window.

- **Copy** (`Ctrl-d`)
  A new frame is created as a copy of the current frame. You are prompted for the name of the new frame. A new frame editing window is immediately opened for the new frame.

- **Invoke New Frame Editor** (`Ctrl-E`)
  You are prompted to click on or type in the name of a frame. Objects that can be clicked on include slot, facet, and annotation labels for which slotunits exist, and values that are names of frames. A new frame editing window is opened to allow editing of the selected frame.

8.6.4 Value Commands

Most of the frame-editing commands you use on a regular basis can be found in this menu. These commands permit you to add, change and delete slot values, facet values, and annotation values. Unless stated otherwise, value can refer to a slot value, facet value, or annotation value, even when only slots are explicitly mentioned. Most of these commands require you to select a value node, although some have reasonable defaults implemented if nodes of other types are selected. Two of the commands require you to select a label (slot, facet, or annotation); if a value is selected instead, the command is applied to the parent of the selected node. Some of these commands do not apply to inherited values: if an inherited value is selected, a warning or a menu of alternatives may be given. If a value input by the user is expected to name a frame, and no frame exists by that name, you may be given the option of creating the frame.

- **Add** (`Ctrl-a`)
  This command takes a slot, facet, or annotation label node. You are prompted for a value to add in a small pop-up window. If the slot takes more than one value, then the new value becomes a new slot-filler for the frame. If the slot takes only one value, then the new value overrides any existing value (including an inherited value) as a slot-filler for the frame.

- **Edit** (`Ctrl-e`)
  A pop-up window appears, containing the old value for editing. If, instead of selecting a value for editing, you select a slot name, then a large pop-up window appears, containing all the slot values for editing. This is helpful if you want to edit all values of a slot at one time.

- **Replace** (`Ctrl-r`)
  An empty small pop-up window appears to accept the new value.

- **Remove** (`Ctrl-k`)
  This value is deleted as a slot value for the frame.

- **Remove All** (`Ctrl-K`)
  This command takes a slot, facet, or annotation label node. All values of this slot are deleted for this frame.

- **Add Facet/Annot** (`Ctrl-A`)
  You prompted for a facet (if a slot is selected) or an annotation (if a value is selected) label, either from a list of possibilities if it exists or in a pop-up text window, and then for a value. If no slotunit for the facet or annotation label exists, you are asked if you want to create one. This operation does not work if facets and annotations are not supported by the frame representation system.

8.6.5 Slot Commands

These commands enable you to create, change and delete slots and slotunits. In addition to slots, they may apply to facet and annotation labels. Be aware that for many frame systems these operations are schema changes, and can affect frames other than just the one currently being edited. They should therefore be used with care. For background information to help you further understand slots and slotunits, see Section 8.3.4.

- **Create** (`Ctrl-n`)
  Prompts for a slot name and creates the new slot. The initial domain of the new slot is the current frame (if a class frame), or the direct parent of the current frame (if an instance frame).

- **Rename**
  Prompts for a new slot name, and renames the slotunit.

- **Destroy**
  Deletes the selected slotunit from the KB. Use this command with caution.

- **Edit** (`Ctrl-E`)
  Recursively edits the selected slotunit in a new frame editing window. This works only for frame representation systems in which slots are represented as frames.

8.6.6 Preferences

A dialog is opened in which you can specify which slots and facets should be displayed, and whether or not inherited values should include where they were inherited from in the node label. The options for which slots and facets to display are

- `All`: show all applicable slots and facets
- `None` (for facets only): do not show facets
- `Filled`: show all slots and facets for which there are values
- `Selected`: show only the designated slots and facets
- `Selected-Filled`: show only those of the designated slots and facets for which there are values

If the options `Selected` or `Selected-Filled` are chosen for either slot or facet display, then upon exiting the dialog, you are prompted to select from a menu of available slots and/or facets. Selected preferences apply to this frame editing window and all subsequent ones (until the preferences are changed again).

8.7 The Spreadsheet Viewer

The spreadsheet viewer allows you to edit slot values in the current KB within the context of a spreadsheet. As shown in Figure 8.4, the columns correspond to slots and the rows to frames. The cells in the sheet are filled with the corresponding frame slot-values from the KB.

There are three steps to launching the Spreadsheet Viewer. First, select frames you want to examine from the Class Hierarchy Viewer. Then choose the **Spreadsheet Viewer** command from the **Applications Menu**. Finally, use the dialog you are presented with to select which slots to display in the spreadsheet and the order. Frames can be listed in the spreadsheet in class order or alphabetically. They can also be displayed using their pretty names or frame names. The frames, selected slots, and corresponding slot values are imported into the spreadsheet.

The Spreadsheet Viewer does not yet have a full range of editing capabilities. Current limitations include

- No classes
  Only instances are displayed in the Spreadsheet Viewer. When you select a class frame from the Class Hierarchy Viewer, its instances are displayed in the spreadsheet. Selected instance frames are themselves displayed.
- No multiple-valued slots
  The Spreadsheet Viewer does not address multiple values. In the case of multi-valued slots, one arbitrary value is chosen from the list of values that exist for a given multi-valued frame/slot cell in the spreadsheet.

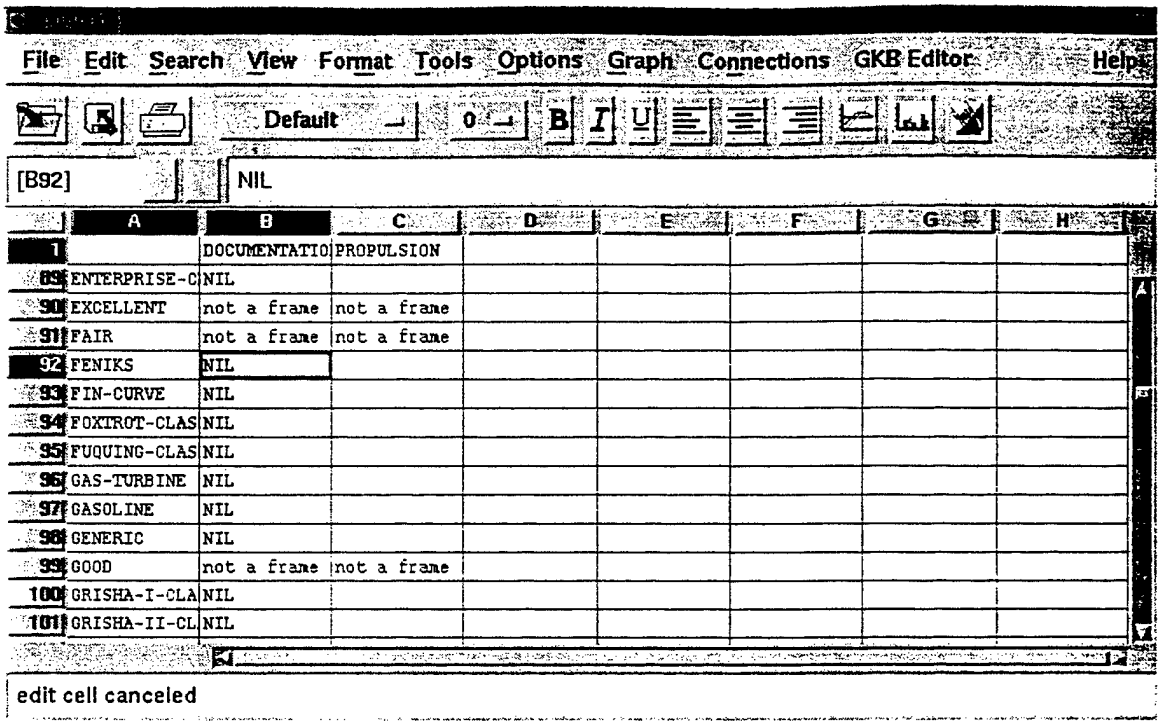


Figure 8.4: A sample spreadsheet view.

- No facets or annotations
  The Spreadsheet Viewer does not display facets or annontations.

- Frame creation but no slot creation
  The Spreadsheet Viewer allows you to add frames, but not slots to the KB. When the data in the spreadsheet is exported back to GKB-Editor, any extra slots in the spreadsheet are ignored. Any frames that have not been added to the spreadsheet explicitly with the menu command are also ignored. Upon exiting the Spreadsheet Viewer, you are notified about the frames that have been created, the frames that have been modified, and the slots that have been ignored.

8.7.1 NExS Spreadsheet

The NExS spreadsheet, a commercial product, is the basis of the Spreadsheet Viewer. You must obtain this software to use the spreadsheet viewer (see the GKB-Editor web pages `http://www.ai.sri.com/~gkb/` for details). For now, all the original NExS Spreadsheet commands are available, but we are working to remove those NExS commands that are not relevant in the context of the GKB-Editor.

8.7.2 GKB-Editor Menu

The **GKB-Editor Menu**, within the NExS spreadsheet, implements the GKB-Editor Spreadsheet Viewer. Commands under this menu support the interaction of the NExS spreadsheet with the GKB-Editor.

- **Add Frame(s)**
  Adds new frames to the current KB. You are presented with a dialog that asks for the name of the frame that you want to add to the spreadsheet. When you type in the name of the frame, the system responds by telling you whether or not that frame can be added to the KB. If it can, the name appears in the list of frames to be added to the KB; if not, a message appears letting you know what the problem is. There are two buttons to the right of the list of frames. **Delete** and **Edit**. When you select a frame from the list, the buttons are activated. Choosing **Delete** removes the frame from the list of frames that will be added to the KB. Choosing **Edit** removes the name from the list and makes it reappear in the box above for editing. When you are done, the edited frame name appears in the list.

  The frames that constitute the list when the dialog is exited are sent to the spreadsheet. The displays in GKB-Editor are updated to reflect the changes when the Spreadsheet View is exited.

- **Save to KB**
  Saves changes in the spreadsheet to the current KB without exiting NExS. The open windows in the GKB-Editor are updated to reflect these changes.

- **Revert Sheet**
  Executing this command reloads the information from the KB into the spreadsheet. This does not mean that the KB reverts to the way it was when the viewer was invoked. Whatever changes were saved to the KB, since invoking the Spreadsheet Viewer, remain in effect. This command simply flushes those changes made in the spreadsheet that were not saved to the KB.

- **Reorder Slots**
  Reorders the columns in the Spreadsheet Viewer by slot. You are presented with the list of slots presently displayed in the spreadsheet viewer. Reorder the list by dragging the slot names up and down. The first slot name corresponds to the left-most slot name in the spreadsheet. When you are finished reordering the list, choose "OK": in response, the columns in the spreadsheet are rearranged to reflect the new order.

- **Close View**
  Exits the Spreadsheet Viewer without saving changes to the KB. An additional dialog appears to confirm exiting the viewer.

- **Close & Save**
  Upon exiting the spreadsheet viewer, the data from the spreadsheet is saved to the KB. You are informed of the frames that were added to the KB, frames that were ignored, slots that were ignored, and frames that were modified.

8.8 Limitations

New users should read the troubleshooting tips in Appendix A. These tips deal with frequently asked questions.

Some known problems are described here. Some of these are fundamental limitations of the Generic Frame Protocol or the various underlying frame systems, and cannot be fixed until those problems are addressed. Others may be fixed in future versions of the GKB-Editor.

- There is minimal constraint checking. If the underlying FRS does constraint checking, then the GKB-Editor may allow an operation that the FRS forbids, which can cause various problems, depending on how the violation is handled by the FRS. For example, the FRS may break to the Lisp debugger, frames may be marked incoherent, or the GKB-Editor may change its display to reflect changes that never occurred. Even if the violation is handled correctly, there may be little or no feedback to you as to why the operation failed. If the FRS does not do constraint checking, then operations may be allowed that nevertheless violate KB constraints. To address this limitation, we would need either (a) better communication between the FRS and the GKB-Editor (through GFP) about which operations are allowed, and whether or not an operation has succeeded, or (b) a mechanism by which constraints could be expressed using GFP and taken advantage of by the GKB-Editor in a uniform fashion.

- There is no way to enter such things as methods for computed slots, complicated constraints that cannot be expressed using simple facet values, or enumerated sets (other than with the :one-of value-type). There also should be a more intuitive graphical way to enter :and, :or, and :one-of value types.

- Most of the editing commands in the relationships editor do not work (and may cause problems) if they are attempted while browsing slot types rather than slot values.

- Facet and Annotation commands may break if these operations are not supported by the FRS.

- There is no existing way to edit the list of KBs in the Knowledge Base Open menu, for example, removing a KB that is no longer in use, or changing the KB package.

- The Kill Ring does not work in all situations. For example, it cannot be used in Motif text gadgets. It also cannot handle strings with embedded newlines (currently, we replace such newlines with spaces).

- Many of the gadget-based dialogs break or look awkward under Lucid CLIM 1.1 (which does not implement gadgets).

- Under CLIM 1.1, we sometimes get residual display ghosts that we think are CLIM bugs. Most of these go away if you bury the window and then bring it to the top again.

- Renaming an instance frame in Loom can cause problems (since that operation is not supported by Loom) for frames that were linked to the frame under its original name.

- Completion does not work properly in all cases. This is a bug.

- When working with Loom, the relationships viewer does not expand slots (relations) with a domain of Thing (or no specified domain).

- Various operations in the Frame Editor and the Relationships Viewer may break or produce incorrect results with FRSs that do not implement slotunits (such as Sipe).

Chapter 9

Summary

The GKB-Editor is a generic editor and browser of knowledge bases and ontologies. It is portable across several FRSs. This is possible because the GKB-Editor performs all KB access operations using the GFP. To adapt the GKB-Editor to a new FRS, one need only create a GFP implementation for that FRS — a task that is usually considerably simpler than implementing a complete KB editor.

The GKB-Editor contains several relatively advanced features, including four 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. It is in active use at several sites. While there is more to be done, the current implementation goes a long way toward living up to its original design goals. It also represents the achievement of a milestone in software reuse for knowledge-based systems, the development of a knowledge-base browser and editor that is reusable across several different knowledge representation systems.

Appendix A

Troubleshooting

Remember that the GKB-Editor is still under development! Although most operations should work correctly when you do the right thing, it is not hard to get into a broken or wedged state. For example, if you are using Loom as your underlying FRS, and you perform an operation that violates a Loom constraint or causes a Loom or GFP error, there may not yet be a graceful way to get out of the resultant inconsistent state, especially if the GKB-Editor thinks a command has succeeded when it in fact failed, and it tries to update the display accordingly. If you find yourself thrown into the debugger in such a situation, first try to return to the GKB-Editor command or top level, and fix the problem by using GKB-Editor commands. If you have some in-depth knowledge of the underlying FRS, you can also try to fix the problem while in the debugger. (Sometimes after returning from the debugger, the interface does not respond to pointer clicks. Press Control-z and try again.) If the problem is not that the KB itself is inconsistent, but that the current graph's view of the KB doesn't match the state of the KB, try beginning a new browse: this will eliminate the current graph's view of the KB, and construct a new one. If these attempts fail, probably the best thing you can do is to revert to the saved version of the knowledge. For this reason, it is important to save your work frequently! Finally, if this fails to correct the problem, you may need to quit and restart the application and/or the Lisp process.

Q. *Why are all the commands in the command menus disabled?*
A. If there is no current KB (i.e., if you just started running and have not opened a KB yet, or you have recently closed or destroyed the current KB), the frame-editing and viewing commands will all be disabled. Commands that should still be available to you are New, and Open in the bf Knowledge Base Menu, and Quit and Kill in the **Application Menu**. Once you open a KB, the other commands should become active. If you have just created a new KB, there will be no frames to view or manipulate. In this case, the only additional command to be enabled is Create Class from the **Frame Menu**. Once you have created a single frame, the other commands should become active also.

Q. *I changed a slot value while in the Frame Editing Viewer, but the new value isn't showing up in the relationships graph. Why is that?*
A. You may be having package problems. If your current package is different from the package your frames are in, then the new value you typed may not be interpreted as corresponding to an actual frame, and therefore will not show up in the relationships viewer (only values that are frames show up in the relationships viewer). To ensure that you get the correct frame in this case, when you enter the new value, preface it with the appropriate package name (e.g., `sipe-cl::army` rather than just `army`).

Q. *I'm using Loom. I created a new slot while in the Frame Editing Viewer, added a value, and exited the Frame Editing Viewer. I have just started editing the same frame again, and notice that the value I added is missing. What happened?*
A. When you create a new slot, Loom doesn't know what the slot range is supposed to be, so Loom assumes that the value you added is supposed to be a frame, unless the value is obviously something else. You can tell that Loom is expecting a frame because when you enter a value that is not a frame, you are asked whether or not you want to create the corresponding frame. If no frame corresponding to the value you specified exists (or gets created), then the value is rejected. The frame editor doesn't know the value was rejected (our bug), so it goes ahead and displays the value. However, the next time you try to edit the frame, the value will be missing. Possible ways to prevent this situation from occurring are

- If you mean the value to be the name of the frame, make sure the frame already exists or gets created.
- If you mean the value to be a symbol or string, you must quote it (for a symbol) or surround it with double quotes (for a string).
- Edit the slot specification, giving a value for the facet `:SLOT-VALUE-TYPE` of either `SYMBOL` or `STRING`. If you do this, you don't need to use quotes when entering new values: the values are automatically interpreted correctly.

You can tell whether or not you need to quote your values by looking at the prompt in the pop-up window. If it says to enter a string or symbol, then you know that your value will be interpreted as the appropriate type, even without the quotes. If it says to enter a form, then Loom doesn't know what type to expect, so you need to use the quotes.

Q. *I'm using Loom, and when I try browsing (or editing) the relationships graph, I am told that there are no slots to follow. I know my KB has relations in it!*
A. The relationships browser works by looking at the roles of a concept or instance. Normally, when you create a Loom relation using GFP, GFP alters the domain concept to take that relation as a role. If the domain is Thing, however, then the domain concept cannot be altered, so no role is created, and the relationship cannot be detected. If your KB was created other than through GFP, you may run into a similar

problem. even if your relations have domains defined other than Thing. This is a GFP problem for which we currently have no solution (the workaround would be to examine every relation in the KB, which would be too time-consuming). The only thing you can do is to use the frame editor to change the domain of your relations.

Appendix B

GFP for Loom

The GFP was designed to present a uniform interface that masks the details of the underlying FRS. While there are many advantages to this approach, the terminology can be confusing for users who are accustomed to interacting with one particular FRS. The problem can be particularly acute for Loom users, as the Loom style of definitions, with necessary and sufficient conditions, does not map easily into the frame, slot, value, and facet structures that make up the GFP. For this reason, we describe here how the various GFP structures are interpreted as they are applied to Loom knowledge bases, and which parts of Loom are not supported by GFP. The GFP and the GKB-Editor have been implemented and tested with Loom 2.1 using lite-instances.

Classes and slots in GFP map to concepts and relations/roles in Loom, respectively. When we create a GFP slot with a particular domain, we do two things: we create a relation corresponding to the slot, and then we redefine the domain concept to take that relation as a role (this behavior produces results analogous to setting the `:domain-implies-role` feature for clos-instances).

Facets are annotations on slots. The GFP uses facets as a means of defining constraints, restrictions and defaults on frames, slots and slot values. To this effect, a set of *standard facets* has been defined. The GFP also supports a more general use of facets to annotate slots with arbitrary values, but this use has no corresponding expression in Loom. The standard facets currently cover only a small number of the possible restriction types available to Loom users, and unfortunately there are some quite common restrictions that cannot be expressed using the GFP (for example, it is impossible to state using GFP that some value should be less than a given number). For this reason, the GKB-Editor also offers a means by which the user can directly edit the Loom definition of a frame, without going through the GFP. We hope this measure will be temporary.

The standard facets available in GFP, and their corresponding interpretations in Loom, are described below. In keeping with Loom behavior, facets can be defined

on classes and slots only – instances can inherit facet values, but cannot have any of their own values.

- **:value-type**
  This facet is a means of specifying a type restriction on the values of a slot. When applied to a slot in a class frame, this facet produces a restriction on the corresponding concept of the form (`:all` *slot type*). When applied to a slot independently of any class, the type becomes the range of the relation.

- **:cardinality**
  This facet specifies the exact number of possible values that a slot may take on. When applied to a slot in a class frame, this facet produces a restriction on the corresponding concept of the form (`:exactly` *N slot*). If this facet is present in a slot independently of any class, the only value it is permitted to take on is 1. The corresponding relation will then be defined with the `:single-valued` characteristic.

- **:maximum-cardinality**
  When this facet is applied to a slot in a class frame, it produces a restriction of the form (`:at-most` *N slot*). This facet does not apply to slots independently of any class (we could have allowed it to have a value of 1, with the same meaning as the :slot-cardinality facet, but we chose not to have two different facets with the same meaning).

- **:minimum-cardinality**
  When this facet is applied to a slot in a class frame, it produces a restriction of the form (`:at-least` *N slot*). This facet does not apply to slots independently of any class.

- **:template-values**
  This facet is meant to be used to specify values that are inherited but cannot be overridden. When this facet is applied to a slot in a class frame, it produces a restriction of the form (`:filled-by` *slot value(s)*). This facet does not apply to slots independently of any class.

- **:default-template-values**
  This facet allows you to specify default values for a slot in a class.

- **:domain**
  This facet is applicable only to slots independently of any frame, and allows the domain of the relation to be specified.

- **:inverse**
  This facet can have a single value, the relation that is the inverse of the slot.

- **:numeric-minimum**
  When this facet is applied to a slot in a class frame, it produces a restriction of the form (>= *slot N*).

- **:numeric-maximum**
  When this facet is applied to a slot in a class frame, it produces a restriction of the form (<= *slot N*).

- **:documentation**
  This facet is applicable only to slots independently of any frame, and contains the documentation for the Loom relation.

Loom does not permit easy incremental modification of defaults, restrictions, and superconcepts for its concepts. The GKB-Editor and the GFP, however, specifically encourage building up frames incrementally. As an implementation detail, when a class is incrementally modified by a GFP operation, the Loom concept definition is retrieved and altered, and the concept is completely redefined.

The GFP currently has no support for Loom productions, actions, or methods.

Bibliography

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

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

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

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

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

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

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

[8] P. Karp, M. Riley, S. Paley, and A. Pellegrini-Toole. EcoCyc: Electronic encyclopedia of *E. coli* genes and metabolism. *Nuc. Acids Res.*, 24(1):32–40, 1996.

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

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

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

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

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

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

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

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

[17] B. Porter, J. Lester, K. Murray, K. Pittman, A. Souther, L. Acker, and T. Jones. AI research in the context of a multifunctional knowledge base project. Technical report, University of Texas at Austin, 1988.

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

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

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

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.