SRI International

The Grasper-CL™ Graph Management System

Technical Note No. 521

January 20, 1993

By: Peter D. Karp, Computer Scientist
John D. Lowrance, Program Director
Thomas M. Strat, Sr. Computer Scientist
David E. Wilkins, Sr. Computer Scientist

Artificial Intelligence Center
Computing and Engineering Sciences Division

Submitted for publication to the journal *Lisp* and *Symbolic Computation*

This work was supported in part by the Defense Advanced Research Projects Agency under the DARPA/Rome Laboratory Planning Initiative under Contracts No. F30602-91-C-0039 and F30602-90-C-0086.

1 Introduction

Graphs are virtually ubiquitous in programming applications. Moreover, graph-structured information is especially prevalent in AI applications, and in the COMMON LISP system itself. We can enhance programs that manipulate graph-structured information by providing these programs with graphical user interfaces that draw graphs, and that allow users to interact with drawings of graph nodes and edges. Therefore, it follows that a programming tool that supports the construction of graph-based user interfaces is a desirable component of a modern COMMON LISP programming environment.

Grasper-CL[1] [5] is a COMMON LISP system for manipulating and displaying graphs, and for building graph-based user interfaces for application programs. The system represents a significant advance over previous COMMON LISP graphers because each level of the Grasper-CL architecture — from the core graph data structures to the interactive display module — has been fully developed and articulated, and is accessible to application programmers. We call this system organization an *open architecture*. In our experience, several different classes of graph-based user interfaces exist. For example, one class produces static drawings of graphs, whereas another class requires extensive user interaction with graph drawings. The open architecture of Grasper-CL supports the development of all classes of interfaces, whereas previous graphers support only one or two classes of interfaces. Grasper-CL graphics operations are implemented using CLIM, the COMMON LISP Interface Manager.

Section 2 of this paper elaborates on the motivations for wanting a system that supports the development of graph-based user interfaces within the COMMON LISP programming environment. Section 3 introduces the architecture of Grasper-CL. Section 4 presents the different classes of graph-based user interfaces. Sections 5 through 8 provide more detailed descriptions of the five levels of the Grasper-CL architecture. Section 10 describes the graph browsing capabilities of Grasper-CL. Section 11 describes applications that have been built on top of Grasper-CL. Section 12 describes previous work on COMMON LISP graphers.

2 Motivations

A graph is a collection of nodes (also called points or vertices) that are connected by edges (also called arcs). Graph-structured information is found in a variety of disciplines, such as reaction networks and chemical structures in chemistry, organization charts in business, circuit diagrams, and family trees. AI programs manipulate a particularly large number of graph types. Since most AI applications are implemented in LISP, a large number of these applications would benefit from a LISP-based system for building graph-based user interfaces. More specifically, AI applications that manipulate the following types of graph-structured information would benefit from Grasper-CL: natural-language parse trees, knowledge representation taxonomic hierarchies, plans, envisionments (in qualitative reasoners), proof trees (in theorem provers and production systems), truth-maintenance structures, contexts (alternative worlds), neural networks, decision trees, and influence diagrams [12].

[1] Grasper-CL is a trademark of SRI International.

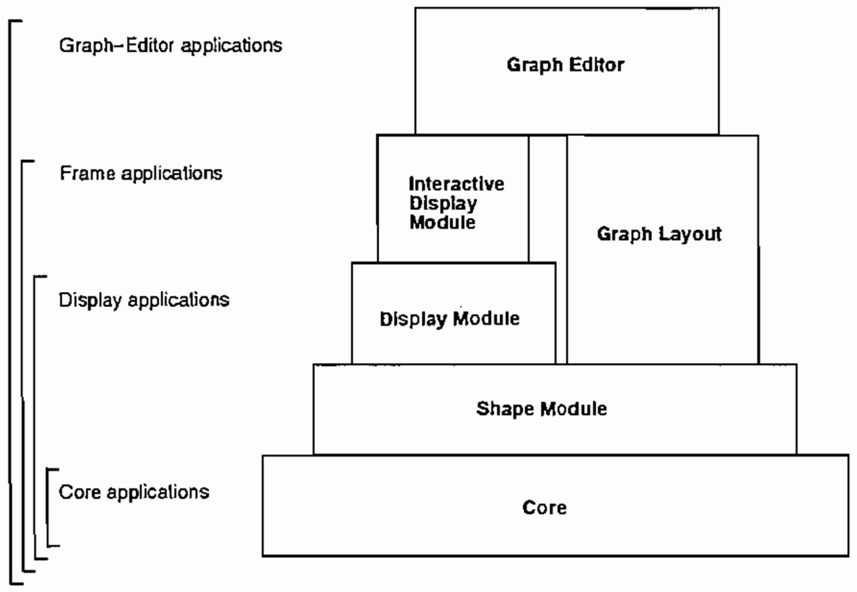


Figure 1: The architecture of the Grasper-CL system.

Furthermore, the COMMON LISP programming environment itself manipulates many types of graphs, so Grasper-CL could see many applications in the COMMON LISP program development environment. Such uses include browsing and manipulation of the dynamic function-call graph produced by a profiler, the static function-call graph produced by a source-code analyzer, graphs of COMMON LISP data structures produced by an inspector, a filesystem directory tree, a CLOS class hierarchy, and a CLOS method hierarchy.

Examples of some of the preceding types of graph-based user interfaces already exist, including graph-based interfaces for knowledge representation systems [1, 6, 11], for planners [16, 18, 17, 15], for contexts [4], for production systems [6], for the CLOS class hierarchy (developed by Franz Inc.), and for the CLOS method hierarchy [7].

3 Architecture

The architecture of Grasper-CL is shown in Figure 1. The Grasper-CL core provides a library of COMMON LISP functions for manipulating a graph abstract data type. These functions perform no drawing operations; they provide a memory-resident graph database facility in which nodes and edges can be created, 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 graphic 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 layout algorithms. Grasper-CL provides several different layout algorithms that produce completely different styles of graph layout, such as a tree or a rectangular layout, and 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 user 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 1 is documented and available to applications. The left side of Figure 1 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 wish 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 inter-city 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 wish to display the final solution computed by our traveling salesman algorithm in a single CLIM 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 cites, 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.

Frame 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. Frame applications execute in the context of a CLIM application frame (hence the name). They can therefore include an arbitrary number of display panes, one or more of which contains the display of a graph. One of the display panes might contain a menu of commands that can be applied to the traveling salesman graph. For example, 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-based applications are a restricted type of frame application — restricted in the sense that the style of user interaction shares much in common with the style of the Grasper-CL Graph Editor, such as the layout of display panes. Such applications will be faster to program than frame applications because of the potential for reusing code within the Graph Editor. For example, imagine that we wish 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 will be easy to define such commands by using the procedures from which the Graph Editor is implemented.

4 Classes of Graph-Based User Interfaces

The user interfaces listed in Section 2 employed a variety of past COMMON LISP graphers, but the interfaces suffered a number of shortcomings due to the limitations of those graphers. (This paper does not address the large number of graphers written in other languages, such as C.) Section 12 discusses individual graphers in more detail, but generally speaking, most previous graphers operate under the following model:

- As input, the grapher accepts a simple description of a graph to be displayed. One common representation is to encode a graph as a list structure; another typical encoding is a list of root nodes plus a function for generating the children of a node.
- In some cases, the input may include a specification of the appearance of graph nodes and edges. That specification might be global for all nodes and edges in the graph, or it might allow the shapes of individual nodes and edges to be tailored.
- Given the description of the graph, the grapher automatically computes a layout for the graph that positions every node and edge in the shape of a tree. It then draws the graph in a window.
- Usually a limited form of user interaction with the graph is allowed. The grapher may allow the programmer to define functions to be invoked when the user clicks the mouse over graph nodes.

This model of graph-based user interfaces makes many constraining assumptions and does not support all classes of interfaces. These assumptions include:

- Previous graphers assume that once a description of the graph has been passed to the grapher, the application will not need to incrementally alter the set of nodes and edges in the graph, and will not need to query relationships in the graph. That is, the graph data structure manipulated by the grapher is not accessible to the application, whereas many applications will benefit from the ability to compute with a graph abstract data type. Furthermore, graphs are assumed to be ephemeral objects, and typically no means is provided for saving and restoring them to and from permanent storage.
- Different application domains traditionally draw graph nodes and edges in a much wider variety of shapes (visual styles) than previous graphers support, and these applications freely intermix many node and edge shapes within a single graph, which some previous graphers do not allow.

- Previous graphers usually assume that a single style of graph layout (such as tree layout) is sufficient for all applications, whereas different applications traditionally draw graphs in a wide variety of layouts.

- Since the graph layout and drawing are computed for the entire graph at once, there is an assumption that all graphs of interest are small enough that the layout and drawing can be computed quickly. In fact, many graphs of interest are larger than several hundred nodes, and require large amounts of time for layout and display.

- Because the model assumes that graphs do not change incrementally as the user works, most graphers provide little if any support for incremental drawing or erasing of individual nodes or edges. If the graph changes, the entire layout and drawing must be created anew.

- The model does not assist the user in understanding relationships in large graphs; when drawn in their entirety, large graphs are tangled and difficult to comprehend.

- The model assumes a restricted form of user interaction with graphs, where users click on nodes or edges to initiate particular actions or to bring up a menu of commands. Some applications require a richer style of interaction with the user, such as allowing full interactive knowledge acquisition.

5 The Grasper-CL Core

Grasper-CL supports an abstract datatype called a GRASPER-GRAPH, or simply a GRAPH. The Grasper-CL core provides facilities for the construction, destruction, and interrogation of GRAPHs. The core can be viewed as a primitive memory-resident database facility based on the GRAPH. This section states the definition of the GRAPH datatype, and then describes the operations on GRAPHs that the core supports. Our terminology is potentially confusing because the GRAPH (uppercase) datatype is a collection of graphs (lowercase) that are stored together in a single file. A *graph* is a collection of nodes and edges, and is synonymous with the term *space*, which will be defined shortly. Put another way, a GRAPH is a hypergraph.

5.1 Definition of the GRASPER-GRAPH

The following definition of a GRAPH refers to the sample graph in Figure 2:

1. A GRASPER-GRAPH consists of a set of spaces, each of which has a unique name.

2. A space consists of a set of nodes and a set of edges. The nodes within a space may be interconnected in arbitrary ways.

3. Each node has a unique name within a space. $N1$ and $N4$ in Figure 2 are node names.

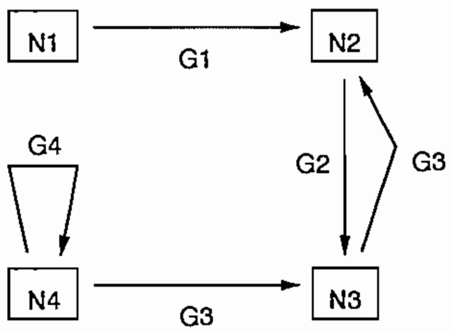


Figure 2: A sample graph.

4. Each edge connects a pair of nodes within a single space. Edges also have names. For example, edge $G1$ connects nodes $N1$ and $N2$.

5. Two edges within the same space need have different names only in the case when they connect the same pair of nodes in the same direction. For example, if we created another edge pointing from $N1$ to $N2$, that edge could not be named $G1$.

6. An edge may leave a node and point back to the same node, such as edge $G4$.

7. Each space has a value.

8. Each node has a value in each space that contains it.

9. Each edge has a value in each space that contains it.

10. The values of nodes, edges, and spaces are made up of tuples. Each value contains zero or more tuples.

11. A tuple is an ordered list of elements. All of the tuples have at least two elements, but may have more.

12. The first element of a tuple is its key. Within any single value, no two tuples have the same key.

The need for spaces may not be self-evident: in our experience many applications profit from the ability to manipulate a set of distinct but related graphs. Each space within the GRAPH abstract datatype holds a distinct graph. For example, a planner might store several related plans in different spaces of a GRAPH.

5.2 Additional GRAPH Terminology

All edges that point away from a node are said to be *outpointing* with respect to that node. $G2$ is an outpointing edge of $N2$. All edges that point to a node are said to be *inpointing* with respect to that node. $G2$ is an inpointing edge of $N3$. $G4$ is both an inpointing and an outpointing node of $N4$. All edges that are connected to a node are said to be *adjacent* to that node. $G1$, $G2$, and $G3$ are the adjacent edges of $N2$. All nodes pointed to by the outpointing edges of a node are said to be outpointing with respect to that node. $N2$ is an outpointing node of $N1$. All nodes from which the inpointing edges of a node originate are said to be inpointing with respect to that node. All nodes connected to the other ends of the adjacent edges of a node are said to be adjacent with respect to that node. $N1$ and $N3$ are adjacent nodes of $N2$.

A pair consists of an edge plus a node. A pair coupled with a space, a node, and a qualifying direction (i.e., outpointing or inpointing) uniquely identifies an edge within a GRASPER-GRAPH. This information is required because edge names are not necessarily unique within a space.

5.3 Core Operations

The Grasper-CL core graph operations are defined as the cross product of six primitive operations with four qualifying directions and with seven types of object specifications, as shown in Figure 3. Some operations in this three-dimensional space are nonsensical. The six primitive operations are Create, Destroy, Set-of, Existence-of, Bind, and Value-of. When combined with a node object specification, we obtain operators such as those for creating and deleting nodes within a space, querying the set of nodes that exist within a space, querying the existence of a particular node within a space, binding value tuples for a particular node, and querying the value tuples of a particular node. The qualifying directions yield operations that apply to either specific (unqualified) nodes, or to all nodes bearing a particular relation to a given node N — the outpointing, inpointing, or adjacent nodes of N. Therefore, one variant of the Destroy operator deletes all nodes adjacent to a particular node; a second variant deletes the set of nodes that are outpointing with respect to a particular node. The other object specifications besides nodes include spaces, edges, pairs, edges-given-a-node, nodes-given-an-edge, and edges-given-an-edge. For example, we can destroy a space, and we can query whether G is an outpointing edge of a particular node. These operators provide a complete, versatile way of managing GRAPHs.

Another set of core operations provides for manipulation of entire GRAPHS for example, to create or destroy a GRAPH, to save a GRAPH to a disk file, and to read a GRAPH from a disk file.

6 The Shape Module

Grasper-CL can draw nodes and edges in a wide variety of visual styles, which the user can control with great precision to model the different graph styles that are used in different disciplines. This section describes the *shape parameters* that control the appearance of a graph drawing, and then presents the mechanism by which a hierarchy of shape defaults can be established.

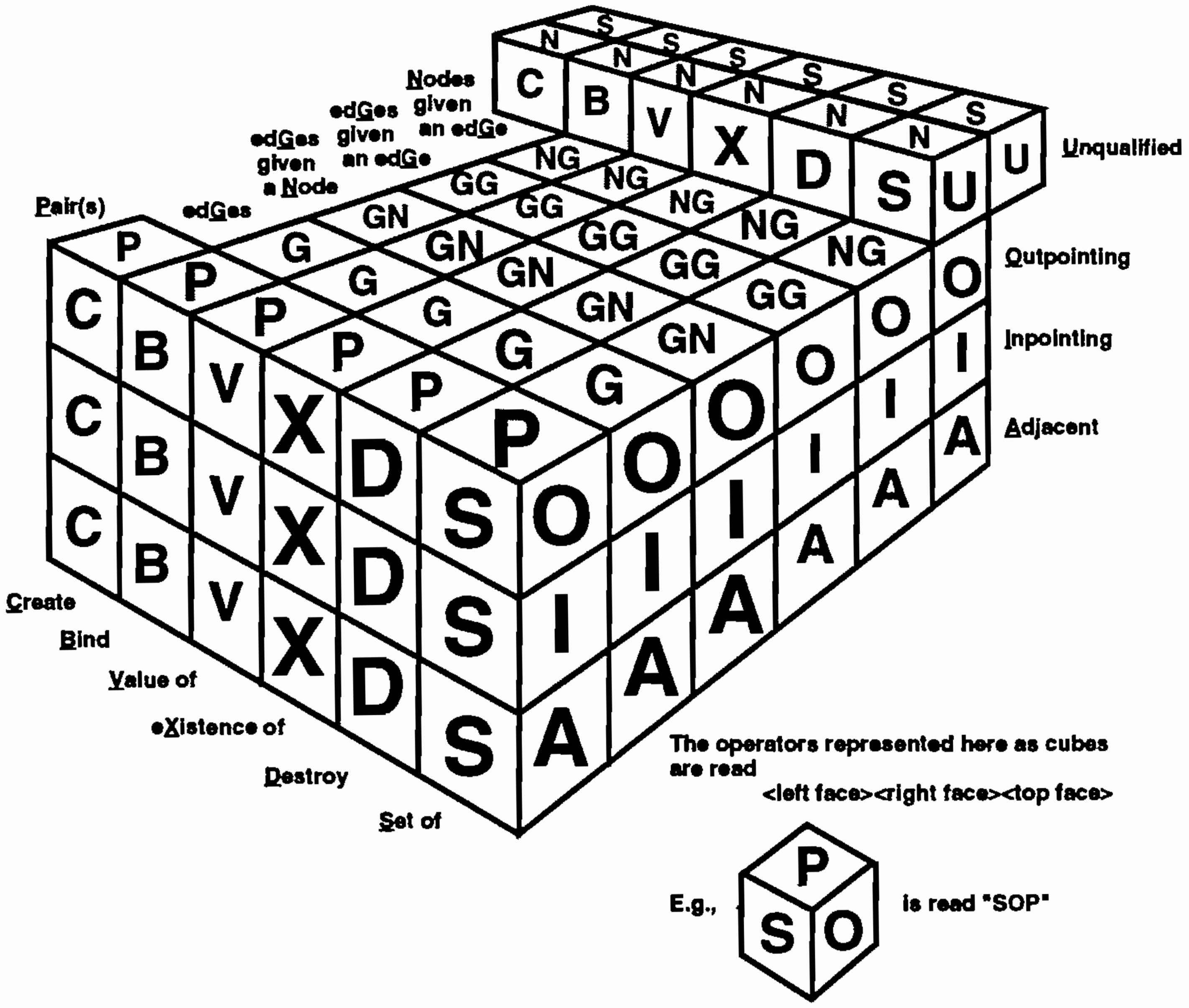


Figure 3: The Grasper-CL core operations. Each point in the three-dimensional space of operations is represented by a single square. The labels on the sides of the square are concatenated to form the name of the operation. For example, the operation to the left of SOP is DOP, which stands for Delete Outpointing Pair.

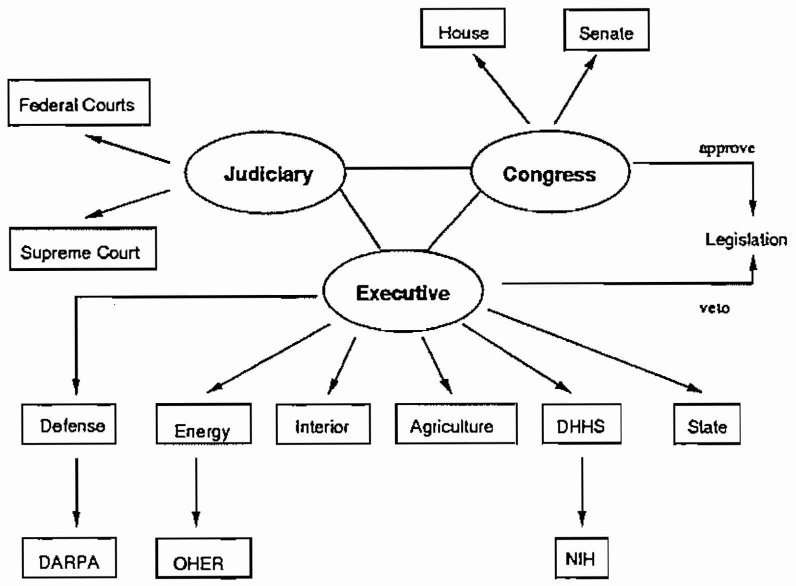


Figure 4: A sample graph showing the partial organization of the federal government.

Figure 4 shows a Grasper-CL graph whose nodes represent entities within the federal government, and whose edges represent relationships between those entities. A node is displayed as an *icon* and a *label,* either of which may be present or absent for a given node. In the node representing the congress, the label is the string "Congress" and the icon is an ellipse. Edges also consist of an icon and a label. The label is not visible for most of the edges in Figure 4, but it is present for the `veto` and the `approve` edges. The edge icon can be further decomposed into the arrowhead, the shaft, and the knot points. Most of the edges in this graph have only two knot points (at the extreme ends of the edges), but the edge between *Executive* and *Defense,* for example, has an internal knot point that gives the shaft of this edge a bend.

The appearance of nodes and edges is governed by a large assortment of *shape parameters.* Each shape parameter controls a single aspect of node or edge appearance. For example, one shape parameter determines whether or not node icons are drawn; another parameter determines what the shape of the node icon is, such as a rectangle, circle, capsule, or diamond; other parameters determine the size of the icon. Shape parameters also control the font of node and edge labels, their location relative to the icon (for example, within the icon or above it), and provide formatting options for multiline labels: the label text itself might be taken from the name of the node or from data tuples stored at the node. Other parameters control the mouse sensitivity of nodes and edges, and the shapes of edge arrowheads — such as the thickness of the edge shaft, whether the shaft is a single or a double line, and whether an arrowhead is drawn. Other parameters determine what action is taken when the user clicks on a node or edge with the mouse. There are 55 edge-shape parameters and 55 node-shape parameters.

6.1 Shape Inheritance and Named Shapes

Shape parameter values can be assigned at a number of different *levels*. At the highest level, every shape parameter has an initial system value that will apply by default to every node or edge within every GRAPH accessed by Grasper-CL. Users can assign new values to these special variables. In addition, a new value can be assigned to any shape parameter at the *graph level,* or at the *space level,* to establish a default appearance for an entire GRAPH or for a single space within a GRAPH. Finally, shape parameter values can be assigned at the level of individual nodes or edges. A value assigned to a shape parameter at any given level is inherited at all lower levels unless the value is explicitly overridden. For example, we might establish at the space level that node icons are to be drawn as circles by default in that space, but override that default for certain nodes in the space. This facility allows us to define different shape parameter defaults for graphs used in different application domains.

Named shapes are named collections of shape parameters. In the traveling salesman application, imagine that the node that is the capital city for a state should be drawn with an elliptical icon, with its label in a large, boldface font. We could create a named node shape called `CapitalCity` that encapsulates these parameter definitions, providing a shorthand, indirect means of creating nodes with that shape.

Taken together, the hierarchy of shape defaults and named shapes provides a powerful way of configuring Grasper-CL to draw nodes and edges in completely different ways for different applications.

The hierarchy of default values for shape parameters can be viewed as an inheritance mechanism for shape parameter values. Because this inheritance occurs at run time, it cannot be implemented using CLOS inheritance. At the system-wide level, each parameter is implemented as a special variable of the same name as the parameter. Default information at the graph and space levels, as well as node and edge-specific information, is stored as value tuples in the respective space, node, and edge; thus, shape information can be manipulated programmatically through the standard core functions. A CLOS implementation would also be inefficient for graphs in which few individual nodes and edges override the default values for more than a handful of shape parameters. In such a case, each CLOS slot that implemented a shape parameter that did not override the default would be consuming additional unnecessary space. Our approach stores the minimum information required to override the default.

Inheritance is computed by a set of macros that retrieve default shape information stored at each level, and establish new bindings for shape-parameter variables whose bindings have changed (using the progv special form). For example, to determine what node icon to use for node `Congress` in space `Government` in the current graph, we would write:

```
(with-graph-shape
  (with-space-shape 'government
    (with-node-shape 'congress
       %node-icon)))
```

The `with-node-shape` macro retrieves shape information local to the `Congress` node, and constructs a `progv` form that creates new bindings for variables that implement the parameters whose values change locally at that node.

Many applications require rapid display of large graphs. Display time is significantly influenced by the node and edge shapes employed. The Grasper-CL documentation contains detailed measurements of display time as a function of node and edge shape to aid application programmers in minimizing display time.

7 The Display Module

The Display Module draws and erases nodes and edges. Functions exist to draw all of the nodes and edges in a given space, and to incrementally draw and erase individual nodes and edges. Drawing can occur at full resolution, or with arbitrary scaling in a *birds-eye view* window (*birds eye* for short). The birds-eye view is very useful for navigation in large graphs, as discussed in Section 10.2.

The display module assumes that the location at which every node is to be drawn has already been encoded by the shape module. The location might have been computed by a layout algorithm, or specified interactively by the user — the source of the node location is irrelevant to the display module.

In contrast, the location of an edge (its start and end points) may or may not be given. When the start and end points of an edge shaft are not given, they are computed automatically at display time. This approach simplifies the design of many layout algorithms by allowing them to position nodes only. In most cases, this approach also frees the user from specifying edge locations interactively. But for some applications, the edge locations computed by the Display Module will not be acceptable, so knot points can be specified by the layout algorithm or by the user.

The Display Module can compute edge locations automatically in three ways. Most commonly, the edge shaft is drawn on the line connecting the centers of the two nodes, and is clipped to the exterior contour of each node, as in part (a) of Figure 5. This approach is aesthetically unacceptable, however, for graphs such as trees, in which we expect to see a horizontal or vertical flow of edges — depending on the directionality of the tree. A shape parameter can cause edges to be drawn with a horizontal or vertical flow. Part (b) of Figure 5 shows edges drawn with a horizontal flow, which is more suitable for this tree. The Grasper-CL layout algorithms prescribe appropriate values for the shape parameters that control edge flow. For example, one of the tree layout algorithms is able to compute both horizontal and vertical tree layouts. When it computes a horizontal layout, it establishes a horizontal edge flow. Henry's system [3] allows each node to determine the appropriate flow for its edges based on the geometry of its edges; this local approach is insufficient when a global flow is required, such as for horizontal tree layout. In Grasper-CL, placement and clipping of edges requires geometric calculations that consume about 10% of the total time required for display (for a graph that has equal numbers of nodes and edges).

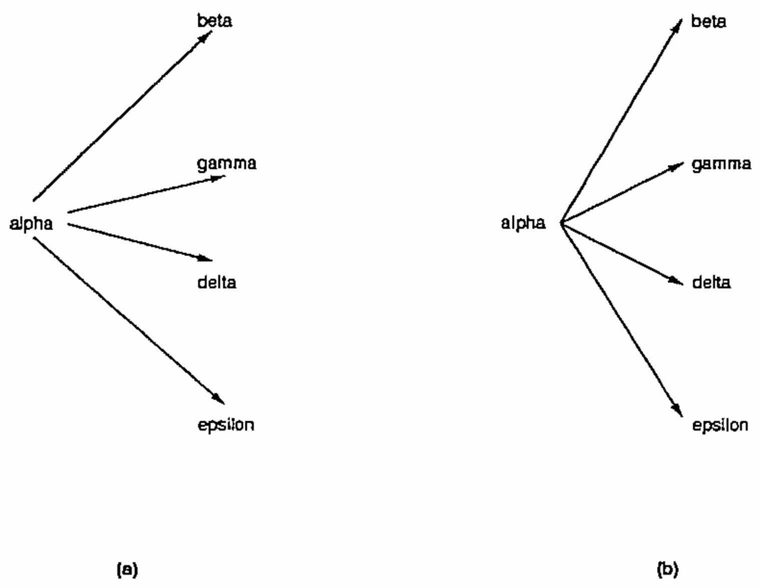
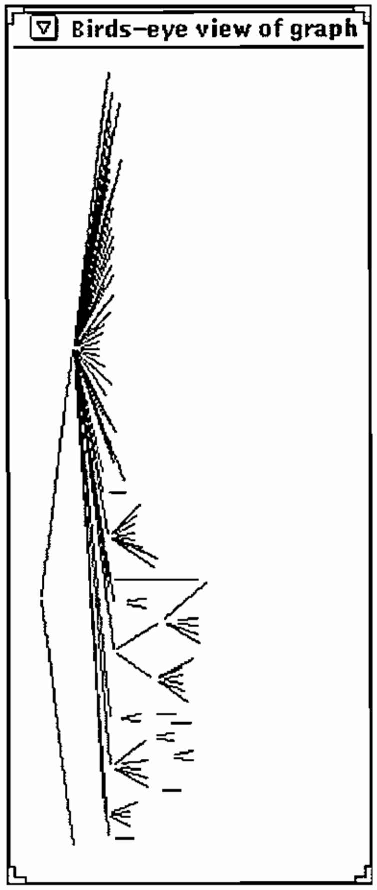


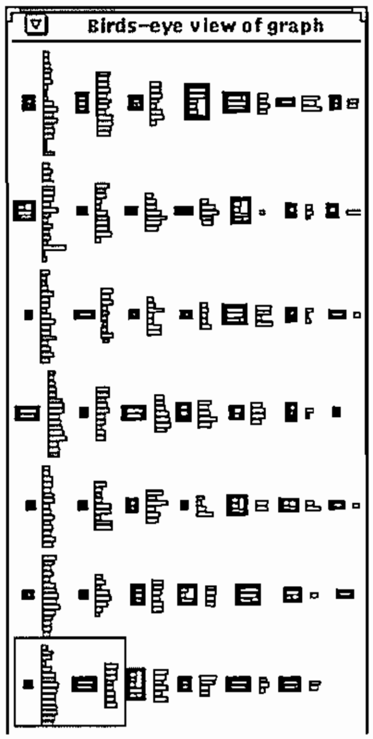
Figure 5: An illustration of automatic positioning based on (a) node centers, and (b) horizontal flow.

Several tradeoffs exist for display of the birds-eye view. First, a space shape parameter allows the user to determine whether only nodes, only edges, or both nodes and edges, are drawn in the birds-eye view (see Figure 6). Different choices have different aesthetic results for graphs with different topologies and layouts. For example, a large graph with a complex tangle of edges is usually easier to understand if the edges are not drawn (Figure 6, right). For more orderly graphs, both nodes and edges can be drawn. Second, the time required to draw the birds-eye view is especially critical when the birds eye is used to drive the browsing of large graphs (see Section 10.2). Node and edge labels are never drawn in the birds eye because the window becomes too cluttered (and because CLIM fonts are not scalable) — saving some time. Normally, node icons and edge shafts are drawn as accurate miniature versions of their full-resolution counterparts. Although this approach provides useful visual clues for distinguishing different classes of application objects (for example, we can distinguish rectangular from circular node icons), if we skip the shape inheritance and geometric calculations and simply draw all birds-eye nodes and edges the same way, display is 60% faster. If we also disable CLIM output recording, we decrease display time to 15% of the original value. This change removes CLIM's ability to refresh the window, and prevents direct selection of nodes and edges in the window; selection can still be approximated if Grasper-CL computes which node is closest to the position of a pointer click rather than using the CLIM selection facilities.

8 The Interactive Display Module

The Graph Editor is a menu-driven Grasper-CL application that allows users to edit graphs interactively. Users can perform a variety of operations on individual nodes and edges, or on





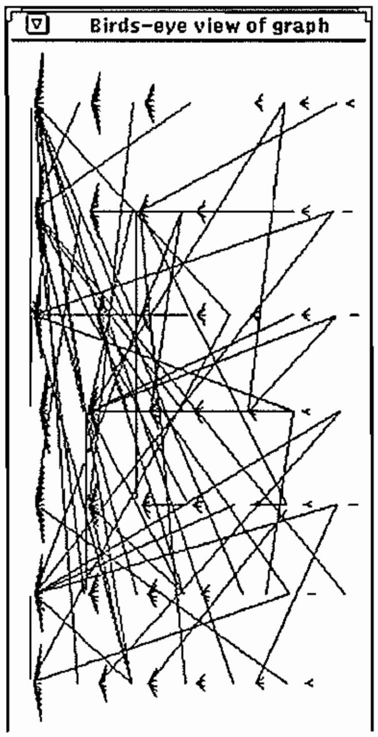
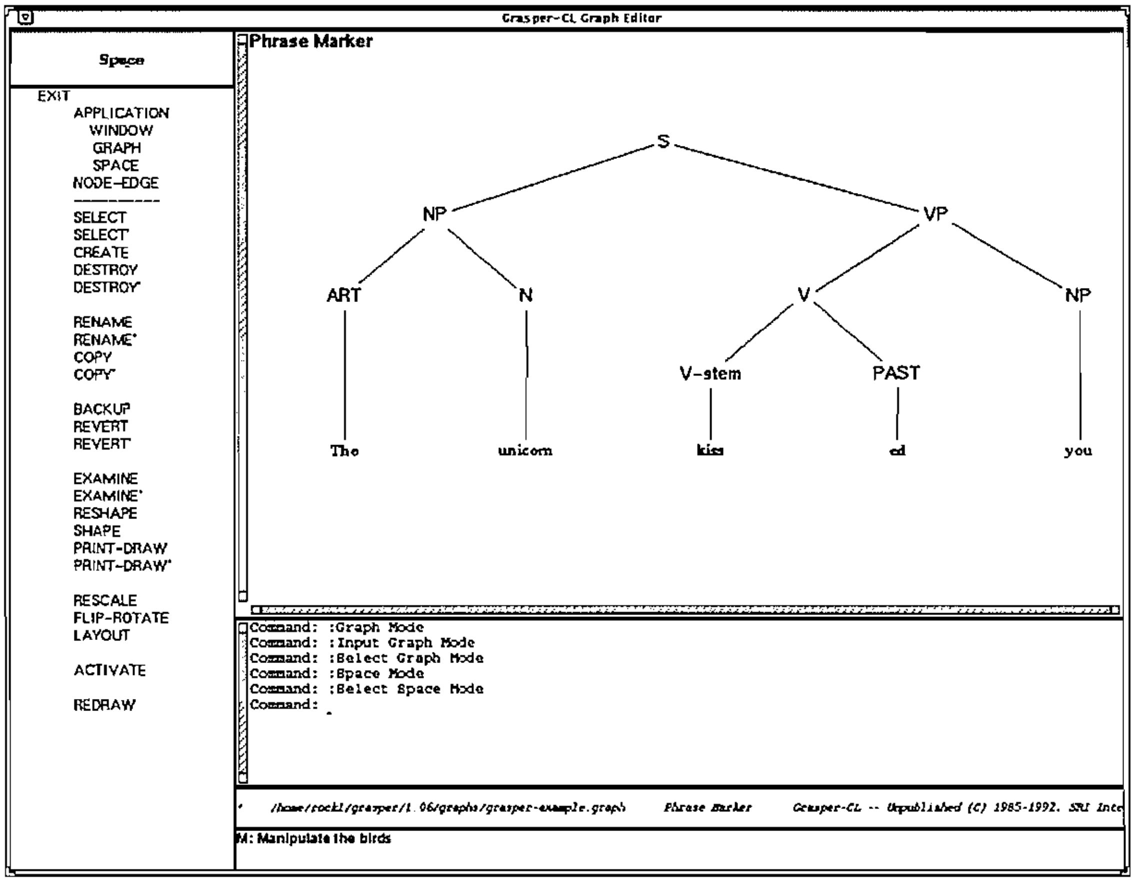


Figure 6: Birds-eye views of three graphs. Depending on the topology of the graph it is sometimes preferable to display edges only (left), nodes only (middle), or both. The rightmost view shows the same graph as does the middle view to illustrate that it is clearer to draw nodes only rather than edges only for this particular graph.

groups of nodes and edges. Those operations include creation, deletion, renaming, copying, moving, examining data tuples, and aligning.

The interactive display module makes two aspects of the Graph Editor programmatically accessible to other Grasper-CL applications. The first is the general command-menu structure of the Graph Editor, as shown in Figure 7. The Graph Editor uses two levels of command menus. The *noun menu* determines what type of object the user currently wishes to operate on, for example, spaces, or nodes and edges. Selection of a noun brings up a lower-level menu of commands that operate on that type of object, such as to delete or rename a space. This noun–verb menu structure is useful in a number of other applications, and can be implemented very quickly. The second is the set of graph-editing operations. Imagine that we want our traveling salesman application to allow users to create, rename, and delete cities through appropriate interactive prompts. Functions in the interactive display module provide just these capabilities, allowing them to be easily incorporated into



```
Command: :Graph Mode
Command: :Input Graph Mode
Command: :Select Graph Mode
Command: :Space Mode
Command: :Select Space Mode
Command: _
```

```
*   /home/rockl/grasper/1.06/graphs/grasper-example.graph   Phrase Marker   Grasper-CL -- Unpublished (C) 1985-1992. SRI Inte
M: Manipulate the birds
```

Figure 7: The Grasper-CL Graph Editor display.

various applications. For example, one function prompts the user to click on a node to be renamed, handles the mouse selection of the node, then prompts for the new name, and then redraws the node with its new name. Additional functions can be created rapidly by modifying the source code for functions in the interactive display module.

9 Graph Layout

Until now our discussions have assumed that the location of each node in a graph display has already been given through some unspecified means. This section describes graph-layout algorithms that assign locations to graph nodes automatically to produce a variety of layout configurations.

The philosophy underlying the Grasper-CL graph-layout capabilities is that there exists no general-purpose graph layout algorithm, either across application domains, or in some cases, within a domain. The reason is that the term "graph layout" actually denotes a large family of related problems. In many cases, and often for historical reasons, different application domains employ graph layouts with radically different appearances. That is, a graph with a fixed topology will often be laid out differently in different application domains. For example, chemical structures, circuit schematics, and family trees are all graphs; their visual presentations are quite different.

Even within a single domain, a single layout algorithm will often be unacceptable, for aesthetic and computational reasons. A single application domain will sometimes include graphs with a variety of topologies. Aesthetically, graphs with different topologies are easiest to comprehend in different layouts; for example, a tree will be easier to comprehend with a tree-structured layout than a circular layout, whereas relationships in a cyclic graph will usually be clearer using a circular layout. Furthermore, a large, complex graph may contain subgraphs of different topologies that, when configured using different layout algorithms, make the overall graph easier to understand.

Different layout algorithms also have different computational properties, and these properties sometimes vary as a function of the topology of the graph. One may be exponential for planar graphs, whereas another algorithm is polynomial. Since different applications have different performance requirements, different algorithms will be appropriate in different situations. For example, in an interactive browser we may accept a less than optimal layout if it is computed quickly, whereas when generating publication-quality output we may be willing to wait much longer for an optimal layout. And since no layout algorithm is likely to be perfect, it is desirable to be able to manually edit an automatic layout, as provided by the Grasper-CL interactive graph editor.

Therefore, our hypothesis is that general-purpose layout services can be provided only by a suite of layout algorithms that have a range of aesthetic and computational properties. Those properties should be matched to the application and the graph at hand. Because of the value of hierarchical layouts of subgraphs of a given graph, it must be possible to compose these layout algorithms. To increase the ease of use of these algorithms, they should also share a common programmer interface to the degree possible.

Previous COMMON LISP graphers provide only a single layout algorithm (usually tree layout), and are therefore inappropriate for many domains and for graphs without tree-like topologies. The remainder of this section describes the common programmer interface for these Grasper-CL layout algorithms, the individual layout algorithms, and the manner in which layout algorithms can be composed.

9.1 The Common Graph-Layout Programmer Interface

We can increase the utility of the layout algorithms by parameterizing them to allow the programmer to customize their behavior in a variety of ways. Although different layout algorithms create very different graphical presentations of the same graph, their operation can be controlled by many of the same parameters, which we call the *shared parameters*. By identifying shared parameters we make the programmer interface to a layout algorithm more uniform. However, the precise interpretation of a given shared parameter may vary somewhat among different algorithms. In addition, most of the algorithms accept additional, idiosyncratic parameters that make no sense for other layout algorithms, which we call *local parameters*. The shared parameters are as follows.

First, every layout algorithm takes as input the name of an existing Grasper-CL space, and a list of nodes, N, within that space. Some algorithms affect all of the nodes in N and only those nodes, whereas other algorithms treat the members of N as seeds and also lay out all nodes connected to nodes in N.

Second, each layout algorithm accepts the following stylistic layout parameters:

- `layout-direction`
  Most layouts have some notion of directionality. For example, a tree layout algorithm can position the root of the tree at the top, bottom, left, or the right of the drawing.

- `node-sort-fn`
  It is often desirable to apply a sort function during the layout process. Different layout algorithms will apply this function in different ways, for example, a tree layout algorithm uses it to sort the children of each node, and the array layout algorithm sorts all of the nodes in the array.

- `vertical-margin, horizontal-margin`
  Consider the bounding rectangle of the set of nodes whose positions were computed by a layout algorithm. The margin parameters specify a translation of the upper-left corner of that bounding rectangle with respect to the coordinate system origin of the graphics window.

- `sibling-h-separation, sibling-v-separation, parent-child-h-separation, parent-child-v-separation`
  For most layout styles it is advantageous to control how close together individual nodes in the layout are drawn. In addition, we often wish to specify different separations for two nodes, depending on the relationships between them. For example, in a tree drawing we might want one separation among a set of sibling nodes, and a different separation between a parent node and its children. Finally, we might want different separations in the horizontal and vertical dimensions. The parameter `sibling-h-separation` specifies the horizontal separation between sibling nodes in a layout in which siblings are drawn next to one another horizontally, whereas `sibling-v-separation` controls the vertical separation of sibling nodes that are adjacent vertically (whether siblings are adjacent horizontally or vertically will often depend on the `layout-direction` parameter).

- `h-alignment v-alignment`
  Some layout algorithms position each node within a cell whose size may be larger than the size of a given node. These parameters determine how nodes are positioned within their respective cells. In the horizontal dimension, a node can be left or right justified, or centered within a cell.

The second part of the shared programmer interface is the set of values returned by each algorithm. Each layout algorithm returns two values: a list of the nodes that were repositioned by the algorithm, and the rectangular bounding box of the resulting layout. For circular layout, for example, the bounding box encloses the entire circle of nodes.

9.2 Layout Algorithms

Our description below of the complexity of each Grasper-CL layout algorithm ignores the time required to evaluate the sorting function if one is employed. The n in the complexity statements is the number of input nodes.

9.2.1 Circular Layout

The algorithm `Layout-circular` positions all of its argument nodes, and only those nodes, along the circumference of a circle. This layout is useful for displaying cycles within a graph, and for drawing graphs with complex topologies. `Layout-circular` runs in $O(n)$ time.

A variety of sorting functions can be used in conjunction with `Layout-circular` to determine the ordering of nodes around the circle. If we sort nodes that have a high number of ancestors towards the top of the circle, an appealing downward flow of edges results (Figure 8(a)). In other cases we wish to order nodes such that edges lie on the circumference of the circle, such as when displaying a graph cycle (Figure 8(b)).

Once nodes have been ordered on the circumference, their actual positions must be computed. We tried one approach to computing node positions that often produced unacceptable results: positioning each node at a fixed angular offset along the circumference of the circle, that is, computing the angle θ_i for node i as $2\pi\, i/n$ radians. The results are unacceptable because nodes near the top and the bottom of the circle sometimes overlap (Figure 9).

Instead, our algorithm splits the nodes into two groups, called A and B, where each group of nodes is to lie along one side of the circle. The diameter of the circle is

$$\begin{aligned} D = max(& \textstyle\sum_{i=1}^{n} height(node_i \mid \; node_i \in \; A) + vspace, \\ & \textstyle\sum_{i=1}^{n} height(nodc_i \mid \; node_i \in \; B) + vspace) \end{aligned}$$

Then, the y coordinate of each node in a group is computed to space the group evenly in the vertical dimension: $y_i = i\,D/2\,n$. The x coordinate is computed to place the node on the circumference of the circle: $x_i = \sqrt{r^2 - y_i^2}$ where r is the radius of the circle.

9.2.2 Array Layout

`Layout-array` positions all of its argument nodes, and only those nodes, in a rectangular array. This algorithm is useful for graphs without a regular topology, and with a relatively low edge density. `Layout-array` runs in $O(n)$ time.

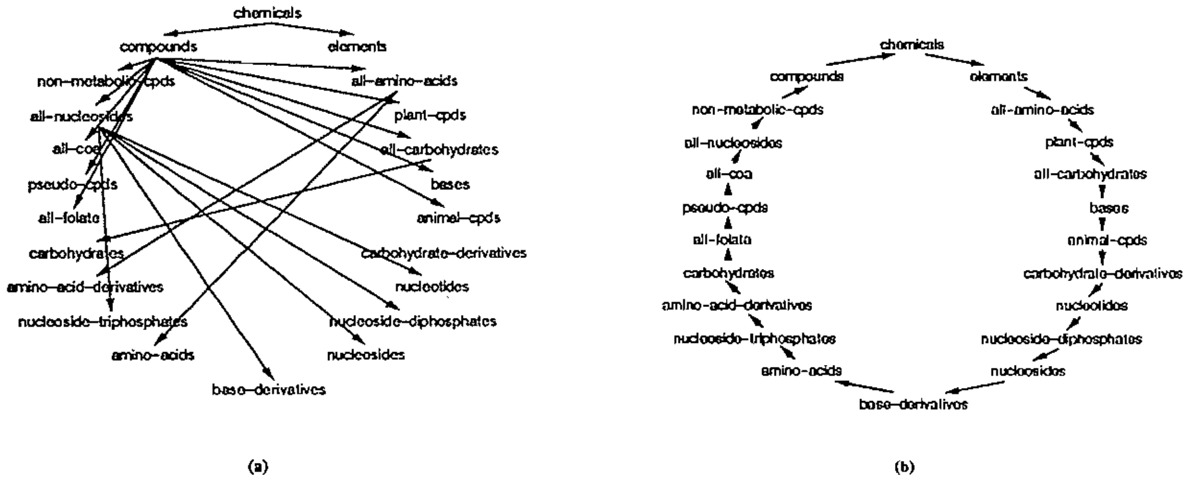


Figure 8: The `Layout-circle` algorithm with nodes sorted to produce downward edge flow (a) and circumferential edge flow (b).

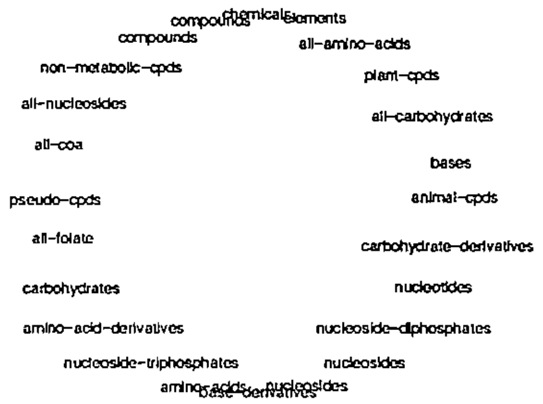


Figure 9: An inferior circular layout algorithm.

9.2.3 Tree Layout

`Layout-tidy-tree` produces a horizontal tree layout in which nodes are packed as compactly as possible, unlike the `Layout-tree` algorithm discussed next. Figure 10 illustrates how the compactness is achieved. Both tree layout algorithms run in O(n) time, so we expect that users will usually prefer to use the tidy-tree algorithm to lay out trees. This algorithm was developed by Moen [10]. `layout-tidy-tree` lays out a set of nodes beginning with one or more root nodes, and progressing to all descendants of those roots. If the set of nodes reachable by edges from the input contains cycles, then those cycles will be broken arbitrarily when laying out the tree. Cycles are "broken" in a figurative sense — all edges in the graph will be drawn in the layout. The point is that this algorithm will work for arbitrary graphs, not just for trees. The resulting layout will be in the shape of a tree, plus, possibly, other extra edges. There is no guarantee as to which edges will fall along the visual tree structure. `Layout-tidy-tree` runs in O(n) time.

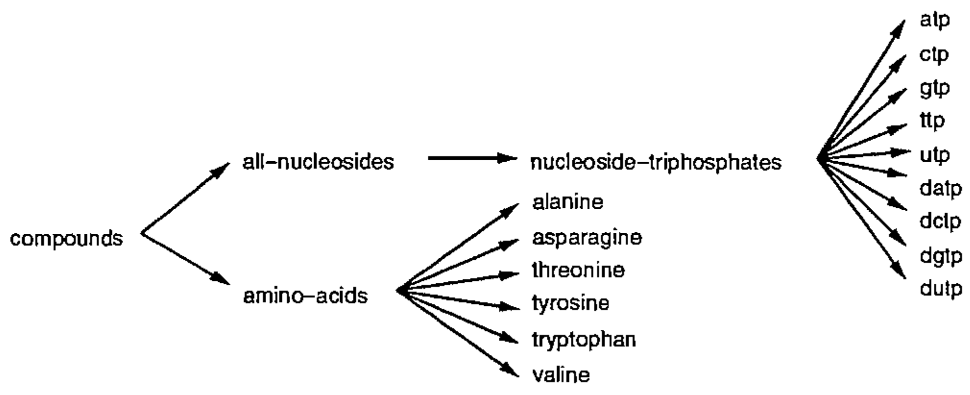


Figure 10: A tree whose layout was computed using `Layout-tidy-tree`. More simplistic tree-layout algorithms would shift the entire `amino-acids` subtree downward such that the top of the `alanine` node was just below the bottom of the `dutp` node. For complex graphs, this space savings allows more information to be visible on the screen at one time.

`Layout-tree` produces tree layouts of arbitrary graphs, and is identical to `Layout-tidy-tree`, with the following exceptions. It is inferior to `Layout-tidy-tree` in that its layouts are not as compact as those produced by `Layout-tidy-tree`, thus the same graph will require more visual space to display. This algorithm is superior to `Layout-tidy-tree` in that it can produce both vertical and horizontal layouts, whereas `Layout-tidy-tree` produces only horizontal layouts. `Layout-tree` runs in $O(n)$ time.

9.2.4 Generations Layout

`Layout-generations` affects the positions of all nodes reachable from the input nodes. Each reachable node N is assigned a *generation number* that reflects the depth of N in the graph. Consider an acyclic graph G such that the longest path in G starts at node A and ends at node Z. The generation number of A in this graph is 0, and the generation number of Z is the length of the path from A to Z. The generation number of every other node in G that is reachable from A is the number of edges lying on the path from A to that node. Generation numbers are computed similarly in cyclic graphs except that cycles are broken arbitrarily when computing longest paths. Put another way, the nodes that are assigned to generation 0 are chosen more or less arbitrarily. The algorithm positions all nodes with the same generation number in the same tier of the layout. The layout consists of a set of parallel tiers that are drawn along a common centerline, as shown in Figure 11. The `layout-direction` determines whether the tiers are horizontal or vertical, and which tier contains generation 0.

The question of how to best position the nodes within a tier is extremely complex. A sophisticated approach will compute an optimal ordering of the nodes within a tier, and will compute optimal node positions given that ordering. Node positions can be influenced by the positions of adjacent nodes in nearby tiers. `Layout-Array` uses a simple heuristic method to order the nodes within a tier, and then spaces the nodes out evenly, and aligns the

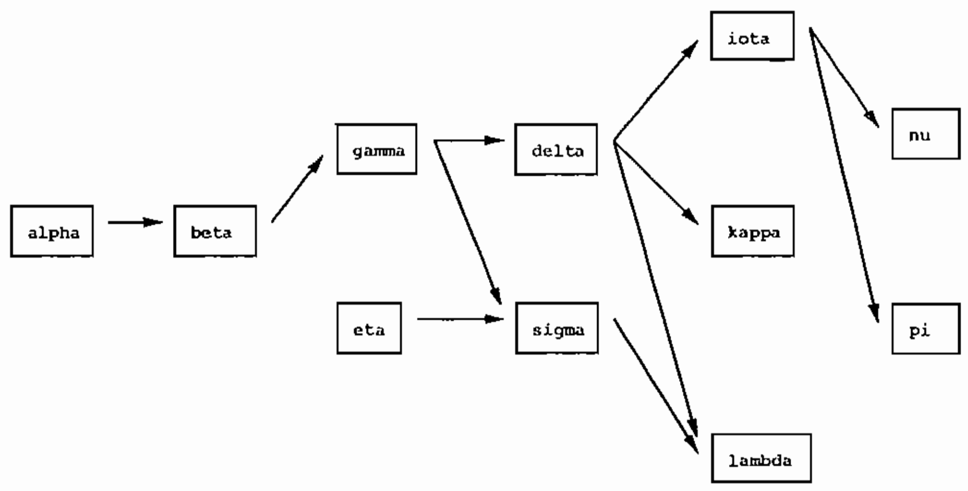


Figure 11: Generations layout.

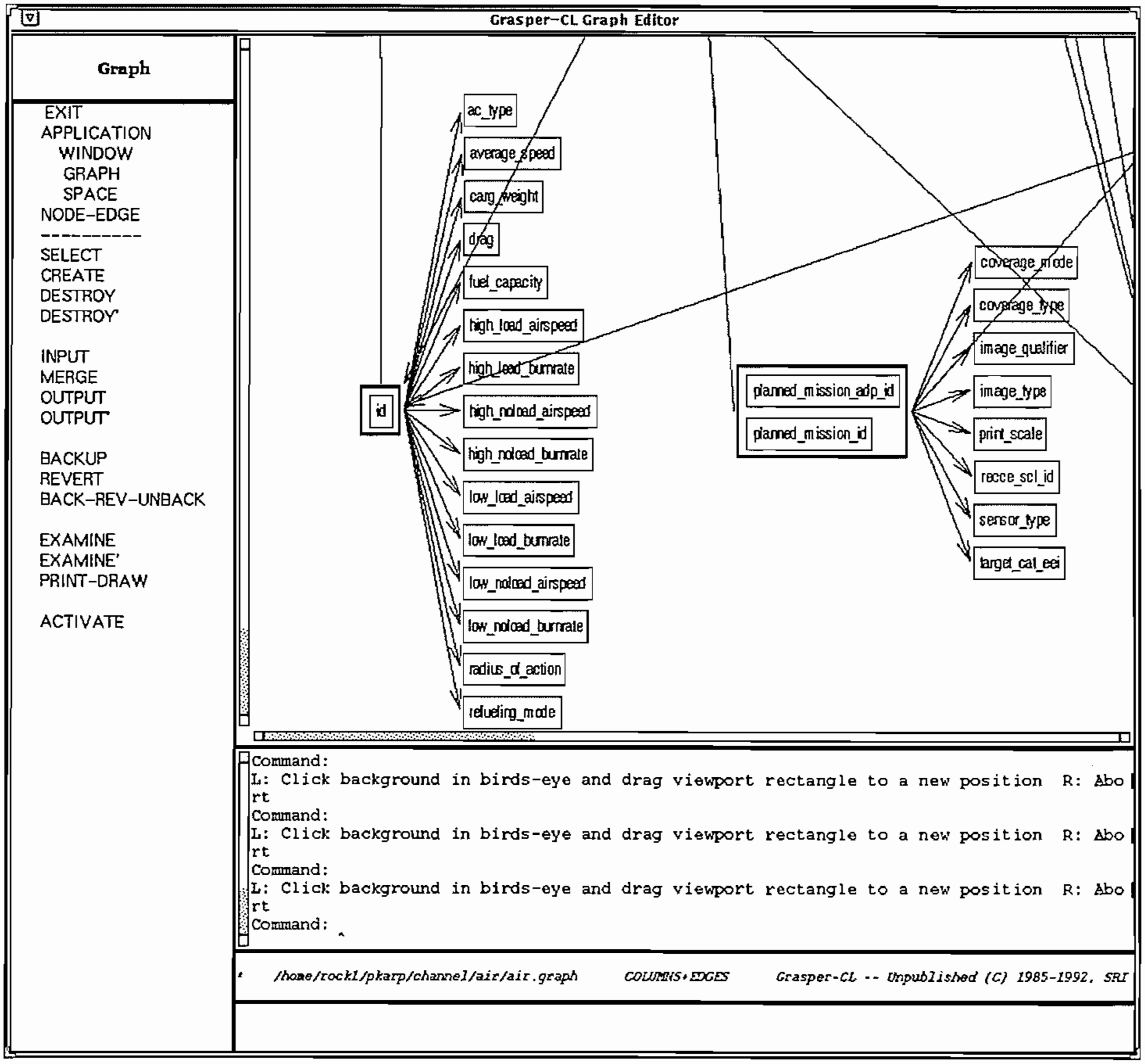
midpoint of each tier with the midpoint of the tallest tier. Messinger provides a thorough discussion of many more sophisticated approaches [9]. `Layout-generations` runs in $O(n)$ time.

9.3 Hierarchical Layout

The challenge of laying out a large, complex graph in an intelligible manner is significant. One approach is to choose a single layout algorithm, such as `Layout-tidy-tree`, and to run that algorithm over the entire graph. A second approach is to divide and conquer the layout problem by laying out subgraphs of a complex graph individually, and to then recursively arrange those subgraph layouts into a larger layout. For example, consider a graph whose nodes correspond to attributes in a relational database schema. Figure 12 shows a three-level hierarchical layout in which primary-key attributes and nonprimary-key attributes are arranged vertically using `Layout-array`, both sets of attributes are placed next to each other horizontally using `Layout-array`, and the entire set of attribute bundles is laid out in a large array (see Figure 6 for a birds-eye view of the schema).

The schema example illustrates one motivation behind hierarchical layout: if we arrange semantically related nodes of a complex graph together spatially, we can make the semantics of the graph easier for a human to perceive. Messinger investigated a second motivation: if we divide and conquer the layout problem, we can increase the performance of layout algorithms that have high (i.e., exponential) computational complexity [9].

Compositional layout involves three subproblems: graph partitioning, layout specification, and layout determination. The graph must be partitioned into subgraphs at some number of levels, that is, the overall graph might be partitioned into subgraphs, some of which are segmented into subsubgraphs. Partitioning can be performed in one of three ways: interactively by the user (as explored by Henry [3]), automatically based on syntactic prop-



```
Command:
L: Click background in birds-eye and drag viewport rectangle to a new position  R: Abo
rt
Command:
L: Click background in birds-eye and drag viewport rectangle to a new position  R: Abo
rt
Command:
L: Click background in birds-eye and drag viewport rectangle to a new position  R: Abo
rt
Command:
```

```
/home/rockl/pkarp/channel/air/air.graph     COLUMNS+EDGES     Grasper-CL -- Unpublished (C) 1985-1992, SRI
```

Figure 12: Hierarchical layout of a portion of a database schema.

erties of the graph, and based on semantic information (for example, semantic information partitions the schema into relations and the relations into attributes). Syntactic partitioning algorithms identify subgraphs with particular syntactic properties, such as high local connectivity (see [9, p130] for a summary of such algorithms).

We propose that a promising area of research is the automatic identification of subgraphs with other syntactic properties, such as cyclic subgraphs and subgraphs that are trees. In particular, we would benefit from the ability to automatically identify subgraphs whose topology is clearly presented by an existing layout algorithm. For example, cycles are easy to comprehend using a circular layout algorithm. Messinger investigated tradeoffs between the syntactic and semantic approaches [9, p130].

Given a graph partitioning, the layout specification indicates what layout algorithm is to be applied to each subgraph, and what layout parameters are used when a layout algorithm is employed. Henry explored the use of a *metagraph* to encode a layout specification, and to allow users to interactively edit that specification [3, p60]. Each node in the metagraph represents a subgraph in the graph partitioning. Connectivity among nodes in the metagraph represents the hierarchy of partitioned subgraphs. Each node in the metagraph contains information that identifies the layout parameters and parameter binding to be applied to the associated subgraph. In Henry's system, the specification is generated interactively by the user. As suggested by our discussion of syntactic partitioning, it may also be possible to generate the specification by syntactically analyzing each subgraph to determine which layout algorithm will present a subgraph most intuitively.

Layout determination computes actual node positions based on a graph partitioning and on a layout specification. It applies the layout algorithms given in the specification, under the specified parameter bindings, to the appropriate subgraph. The subgraph might consist of a set of nodes, or it could recursively contain a set of subgraphs. Therefore, each layout algorithm must be able to position an entire subgraph just as it can position a set of nodes.

Such hierarchical layouts can be obtained for our database schema example by composing two Grasper-CL layout algorithms in the following way. In one Grasper-CL space we would lay out the set of attributes within a relation as a circle. `Layout-circle` returns the bounding box of that circle. We then create a *super node* in a second space with a nonexpandable node icon, whose height and width are the same as the height and width of the bounding box of the circle. This process is repeated for each relation in the first space. We next run `Layout-array` over the super nodes in the second space to assign them to proper positions within an array. Finally, we use the new positions of the super nodes to adjust the positions of their subnodes within the first space, thus moving the circular groups of attribute nodes into the array formation.

10 Graph Browsing

Thus far our discussion has assumed that, like most other graphers, Grasper-CL always draws graphs in their entirety. This approach is cumbersome for large graphs because

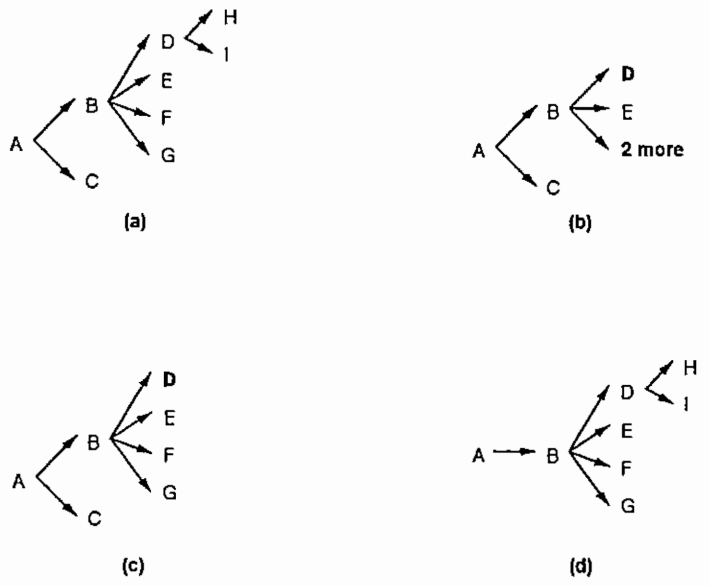


Figure 13: Incremental browsing of a full graph.

drawing time becomes too long. In addition, relationships among nodes and edges become obscured when nodes that are close with respect to connectivity are spatially far from one another, and when a plethora of crisscrossing edges yields a confusing tangle. For these reasons we have developed two browsers that allow users to incrementally explore relationships in a complex graph. One browser is based on the birds-eye view; the other is an incremental subgraph browser.

10.1 Incremental Subgraph Browsing

This browser allows users to incrementally explore local neighborhoods within a complex graph. We call the full-size graph to be browsed the f-graph, and the subtree that is actually displayed the b-graph (browse graph). The local neighborhood is displayed as a tree whose root we call N. A user who invokes the browser specifies N in one of several ways: by clicking on N in the birds-eye view, by clicking on N in the normal-resolution display, by typing the name of N, by selecting N from a menu of all nodes in the current space, or by allowing Grasper-CL to compute the root of the f-graph.

The subtree display is based on a user-defined depth limit and breadth limit. No nodes below the depth limit are displayed in the b-graph, and the display of every list of sibling nodes that exceeds the breadth limit is truncated in the b-graph. For example, imagine that the f-graph to be browsed is that in part (a) of Figure 13, and that the depth limit and breadth limit are both 2. The initial b-graph display is that in part (b). Nodes H and I are not displayed because one must traverse three edges to reach A from H or I, thus exceeding the depth limit; nodes F and G are not displayed because the number of children of B exceeds the breadth limit.

Once the b-graph is displayed, the user can further explore the underlying f-graph. Clicking on the *breadth-cutoff* node whose label is "2 more" will double the number of children of B displayed, thus producing the drawing in part (c). Node D in parts (b) and (c) is a *depth cutoff* — indicated by drawing it in bold face — therefore clicking node D from part (c) would yield the full drawing in part (a). An option forces the browser to limit the complexity of the display when a depth cutoff is expanded by truncating the top portion of the graph. In this mode, expanding D in part (c) would yield part (d) instead of part (a). All subtrees rooted at siblings of A are deleted in this mode. The idea is to treat the depth limit as a harder constraint that does not only have meaning when browsing begins.

In addition, middle clicking on any node removes the subtree below that node from the b-graph display.

The most complex aspect of implementing this type of browser concerns the incremental display of nodes and edges that occurs when a user expands or contracts a region of the b-graph. That is, when the user expands node D in part (c) of Figure 13 to yield part (d), the browser should draw nodes H and I and their inpointing edges only, without redrawing the entire b-graph.

At least three implementation strategies exist for such incremental graph layout and drawing. The simplest is to compute a fixed layout for the f-graph when browsing begins, and to not alter the layout at all during browsing — we alter only the nodes and edges that are drawn at a given moment. Node expansion and contraction operations determine *which* nodes and edges are drawn in the b-graph, but nodes and edges are always drawn at their preassigned position in the f-graph layout. Franz Inc. used this method in a 1992 version of its CLOS class browser. The drawback of this method is that even when only a few nodes are visible, they may be located very far from one another, in which case they are not visible on the screen at once, thereby obscuring graph relationships.

The most complex strategy is to utilize an incremental layout algorithm that automatically alters the b-graph layout to accommodate additional nodes. Moen sketched out such an algorithm [10], but it is very complicated.

A less complex algorithm yields almost equivalent performance. The approach is to *recompute* the layout of the entire b-graph when nodes are expanded or contracted, but to only *redraw* those nodes whose positions changed with respect to the previous layout. This approach utilizes the nonincremental version of Moen's layout algorithm. The speed of this approach is acceptable because the time required to remember the previous node positions, recompute the entire layout, and determine which nodes are not in their previous positions, is small compared to the time required to redraw those nodes and edges that have moved, appeared, or disappeared. The chief disadvantage of this approach is that because the layout is not truly incremental, adding a single node to the layout may radically change the structure of the tree that is drawn, in a cognitively dissonant manner. This situation can occur only when the f-graph is not a tree, but has extra edges. The tree layout algorithm arbitrarily chooses which edges lie along the backbone of the tree, and which edges are "extra." As nodes are added or removed, the edges along the backbone of the tree may change suddenly, which can disorient the user.

A refinement of this algorithm is required because the new layout computed by Moen's algorithm sometimes shifts the entire tree up or down when a new node is added. Therefore, we compute the most frequently occurring (dx, dy) pair across all nodes in the tree, where (dx, dy) is the change in the position of a node due to the new layout. We translate the entire tree by $(-dx, -dy)$ to minimize the number of nodes that must be redrawn. A further refinement is to detect when all nodes in a subtree of the overall layout are translated from their previous position by the same (dx, dy); CLIM provides facilities whereby the drawing of the entire subtree can be translated as a unit much faster than its nodes can be redrawn individually. Without this capability, the time required to redraw individual nodes at their new positions would be unacceptably high (edges coming into the subtree must be redrawn.)

10.2 Birds-Eye View Browsing

The other graph navigation facility of Grasper-CL is based on the birds-eye view. This low-resolution graph display is useful because the entire graph is visible at once, but the birds-eye view is frustrating because information is lost at low resolution. The user can regain some of this information by identifying an individual node through the action of passing the mouse pointer over a node, causing the name of the node to be displayed in a special window. Further, a menu of birds-eye operations allows a user to request the following actions for a given node, N:

- Draw all outpointing, inpointing, or adjacent edges with respect to N
- Erase all outpointing, inpointing, or adjacent edges with respect to N

In this way relationships in the graph can be displayed selectively, and incrementally, without producing tangled drawings.

The user can also employ the birds-eye view to control the scrolling of the high-resolution graph drawing. A rectangle displayed within the birds eye indicates the position of the high-resolution drawing within the entire graph. When the user drags the rectangle, the high-resolution drawing scrolls.

11 Grasper-CL Applications

11.1 SIPE-2: The Anatomy of a Grasper-CL Application

This section describes the workings of a Grasper-CL application — the graphical user interface (GUI) to the SIPE-2 planner — to provide a concrete example of how Grasper-CL is used to construct an interface to a program that manipulates graph-structured information. The SIPE-2 planner generates plans that accomplish problem-solving goals in a variety of domains. SIPE, an earlier version of SIPE-2, had its own graphical user interface that was not built on Grasper-CL. Building a Grasper-based GUI in SIPE-2 provided many significant new capabilities at little or no cost in terms of implementation time. In addition, the new GUI is much easier for the user to understand. The new capabilities include:

- SIPE-2 features that are accessible in command menus
- Powerful ways to customize the appearance of a graph drawing
- The ability to draw every important SIPE-2 data structure for the user (previously, only plans could be drawn)
- Several methods of displaying properties of large plans
- Flow of the planning and execution process are depicted by dynamically updating the graph drawing while SIPE-2 algorithms are running

The SIPE-2 GUI is a graph-editor application, meaning that it makes use of all levels of the Grasper-CL architecture shown in Figure 1. The following subsections describe the use of the Interactive Display Module, the use of layout algorithms, and the ability to follow the execution of SIPE-2 algorithms.

11.1.1 SIPE-2 Command Menus

Figure 14 shows the noun and verb menus used by the GUI. The five nouns let the user choose the type of objects on which the verbs (commands) will operate. The PROFILE noun activates commands for setting defaults that allow the user to customize the behavior of SIPE-2. The DOMAIN noun activates commands that apply to the problem domain as a whole, for example, inputting and inspecting a domain. The PLAN noun activates commands that apply to a specific plan, including executing a plan and solving a problem to produce a plan. The DRAWINGS noun activates commands that can draw plans as well as other important system data structures. Finally, the NODE noun activates commands that apply to specific nodes in the current graph drawing.

The main effort in developing the GUI was naming these commands and deciding how to group them. Most of the code implementing the commands already existed in SIPE-2 or Grasper-CL. All commands in the DRAWINGS menu except *New View* and those under *DRAW*, and all commands in the NODE menu except *Print, Resource, Argument,* and *Predicate* were already implemented by the Grasper-CL interactive display module, thus providing several useful capabilities without additional programming cost. For example, the *Reshape, Move,* and *Align* commands invoke Grasper-CL code to customize drawings. Even the *Predicate, Resource,* and *Argument* commands rely heavily on Grasper-CL capabilities as SIPE-2 simply constructs a list of nodes satisfying a certain condition and passes them to a Grasper-CL function that highlights them. Grasper-CL already had functions for computing all the predecessors and successors of a node and highlighting them. The GUI commands that highlight nodes have proven extremely useful for understanding large

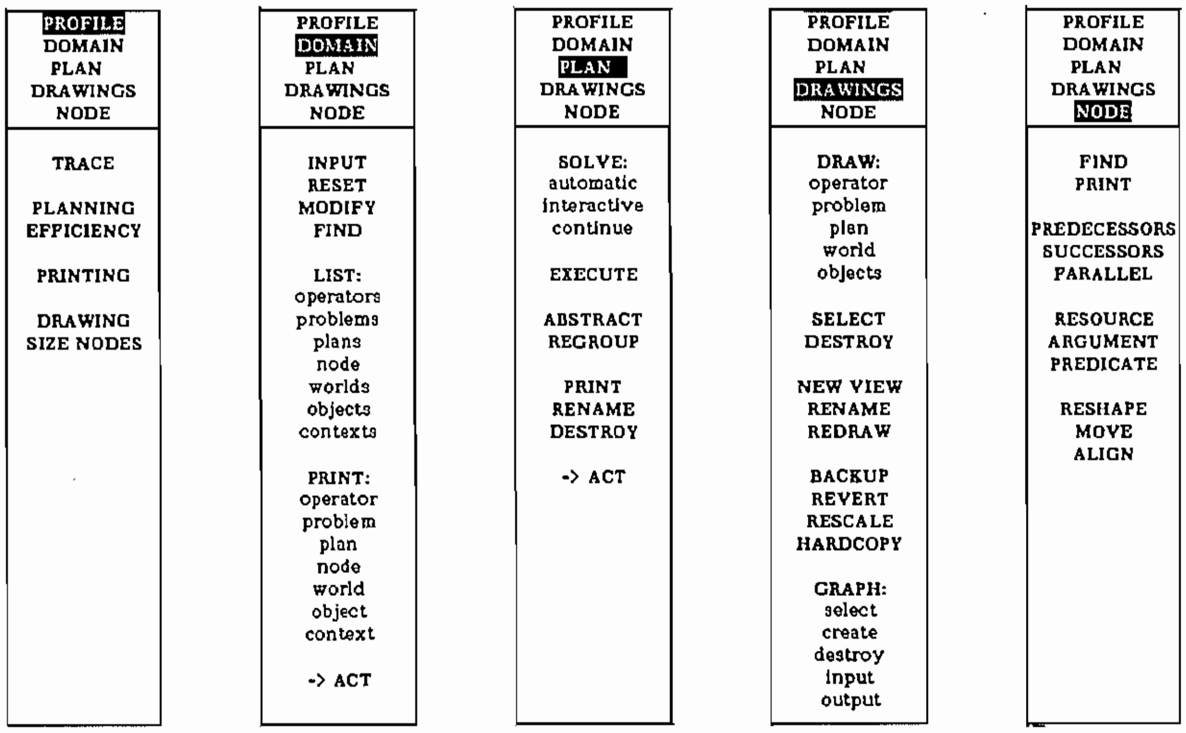


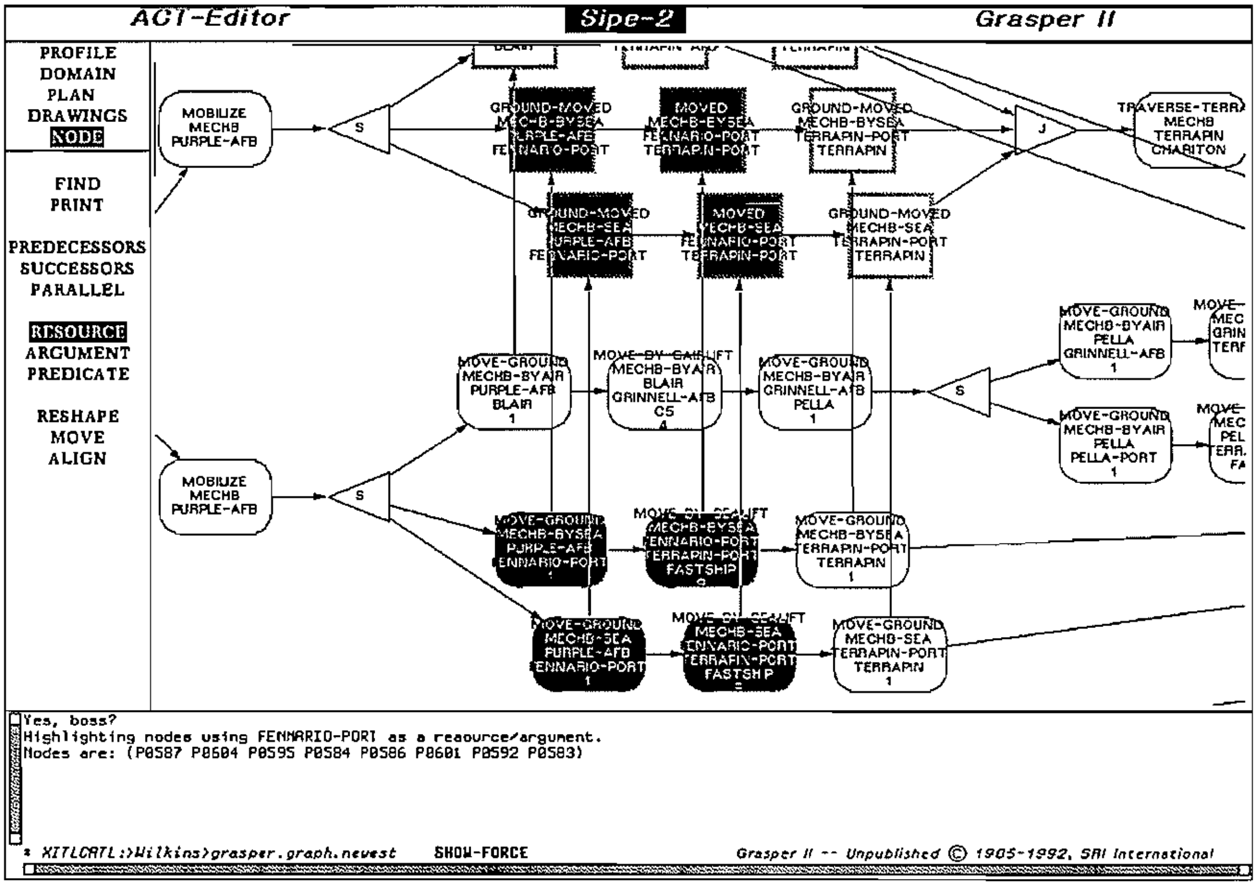
Figure 14: SIPE-2 GUI noun and verb menus. Clicking on each of the five nouns at the top brings up a different verb menu at the bottom.

plans. For example, there may be a couple dozen of nodes out of a thousand that use a particular planning resource, and highlighting these nodes with the *Resource* command helps understand resource usage in the plan.

11.1.2 Drawings

The DRAWINGS command menu drives the static displays of plans, the sort hierarchy (a representation of the object types involved in a planning problem), the plan operators (the planner's representation of actions), and the world model (the assertions that hold about the planning world). When the user selects a data structure for display, the SIPE–2 GUI first translates its internal representation into a Grasper-CL graph, for example, each class in the sort hierarchy becomes a node in the resulting graph. Then an appropriate layout algorithm is called and the resulting graph displayed. `Layout-tidy-tree` lays out the sort hierarchy, and `Layout-array` lays out the world model. As described below, SIPE–2 uses its own layout algorithm for plan drawings. The drawing of operators invokes both `Layout-array` and the plan layout algorithm.

As well as simply scrolling through a plan, the SIPE–2 GUI allows users to visualize a number of different relationships in a plan. The user can view the plan at different levels of abstraction by switching among the multiple Grasper-CL spaces that contain different abstractions of the plan. At a given level of abstraction, the labels of plan nodes can be generated in different ways to show different amounts of information about a node,


ACT-Editor
Sipe-2
Grasper II
PROFILE
DOMAIN
PLAN
DRAWINGS
NODE
FIND
PRINT
PREDECESSORS
SUCCESSORS
PARALLEL
RESOURCE
ARGUMENT
PREDICATE
RESHAPE
MOVE
ALIGN
MOBILIZE MECHB PURPLE-AFB
S
J
TRAVERSE-TERRAIN MECHB TERRAPIN CHARITON
MOVE-GROUND MECHB-BYAIR PURPLE-AFB BLAIR 1
MOVE-BY-CAIRLIFT MECHB-BYAIR BLAIR GRINNELL-AFB C5 4
MOVE-GROUND MECHB-BYAIR GRINNELL-AFB PELLA 1
MOVE-GROUND MECHB-BYAIR PELLA GRINNELL-AFB 1
MOVE-GROUND MECHB-BYAIR PELLA PELLA-PORT 1
MOVE-GROUND MECHB-BYSEA TERRAPIN-PORT TERRAPIN 1
MOVE-GROUND MECHB-SEA TERRAPIN-PORT TERRAPIN 1
FASTSHIP

```
Yes, boss?
Highlighting nodes using FENNARIO-PORT as a resource/argument.
Nodes are: (P0587 P0604 P0595 P0584 P0586 P0601 P0592 P0583)

* XITLCATL:>Wilkins>grasper.graph.newest     SHOW-FORCE          Grasper II -- Unpublished © 1985-1992, SRI International
```

Figure 15: A SIPE-2 Plan Highlighting Resource: Fennario Port

such as the name of the planning operator for an action node, or the operator name plus its arguments. To see a detailed printed description of a plan action, the user can click the mouse on that node. The user can highlight all the successors of a node, or all the predecessors of a node, or all the action nodes that are unordered with respect to a node, or all nodes that mention a certain object or have certain predicates in their effects.

An example illustrates many of these features. Figure 15 shows the SIPE-2 GUI displaying a plan graph from the SOCAP military operations planning domain [16]. The plan was generated by the command *Automatic Solve* from the PLAN menu, and displayed by the command *Draw Plan* from the DRAWINGS menu. The rectangular nodes with rounded corners depict actions that must be executed, the rectangular nodes depict conditions that must be true, and the triangular nodes depict the beginning or end of parallel courses of action. After drawing the plan, the user wanted to see the utilization of Fennario Port in the plan. Figure 15 shows that by using the *Resource* command in the NODE menu, all nodes in the plan that mention Fennario Port can be highlighted.

The entire plan of Figure 15 contains hundreds of nodes, and only a small subset can be on the screen when the nodes have labels. The user can understand the larger picture either by scrolling the viewing window, or by using the birds-eye view facility of Grasper-CL (see

Section 10). Clicking on any of the nodes will produce a printed representation of all the SIPE-2 information actually present at that node.

11.1.3 Layout

SIPE-2 uses its own custom layout algorithm, developed originally as part of SIPE and easily adapted to use with Grasper-CL, to position the nodes in a plan drawing. An example of such a layout is shown in Figure 15. The need for a custom layout algorithm confirms our hypothesis from Section 9 that there exists no general-purpose graph layout algorithm. And although the Grasper-CL suite of layout algorithms handles four of the five types of graph drawings produced by SIPE-2, no existing Grasper-CL layout algorithm properly handles the complex topology of plan graphs. The unique features of these plan graphs are described here.

Plan graphs all have a special topology. SIPE-2 distinguishes between two types of graph edges, which we call *operator edges* and *plan-critic edges.* The former are created when operators are used to expand a plan and the latter are created by planning algorithms, called *critics*, that order actions. When only operator edges are considered, a plan graph has the property that its orderings can be expressed as a string composed of node names and an arbitrarily deep nesting of two sets of brackets. This property is best explained by an example graph that has this property. Such a graph is the blocks-world plan graph drawn by the SIPE-2 GUI in Figure 16. In constructing a string to describe this graph, let us use parentheses for grouping substrings that will be processed in parallel and square brackets for grouping substrings that will be processed sequentially. In a plan graph, Split and Join nodes (labeled "S" and "J" respectively in Figures 15 and 16) represent the parentheses. The ordering edges in Figure 16 are described by the following string:

```
[ P-1 ( [(P57 P58) P59] [(P67 P68) P69] ) END ]
```

We are not aware that this property of graphs has been named, but note that a string representing such graphs can be generated by a context-free grammar. We therefore refer to graphs with this property as *context-free* graphs. A plan-critic edge specifies an ordering between a node from one element of a parallel grouping and a node from a different element of the same parallel grouping. (These nodes may be nested within their respective elements.) The inclusion of even one plan-critic edge makes a context-free graph become non-context-free. For example, in Figure 16 the addition of an ordering edge from any of P57, P58, or P59 to any of P67, P68, or P69 would produce a graph that is no longer context-free.

The SIPE-2 layout algorithm exploits the context-free topology of plan graphs. In particular, plan-critic edges are ignored by the layout algorithm. The operator-edge successors of a split node are placed in a vertical stack with each successor centered in enough space to allow for the maximum height that will be reached over its extent to the join node that matches this split node (using only operator edges). After the nodes have been placed in this way, the plan-critic edges are drawn where they happen to fall — this can produce some clutter in a plan with many plan-critic edges.

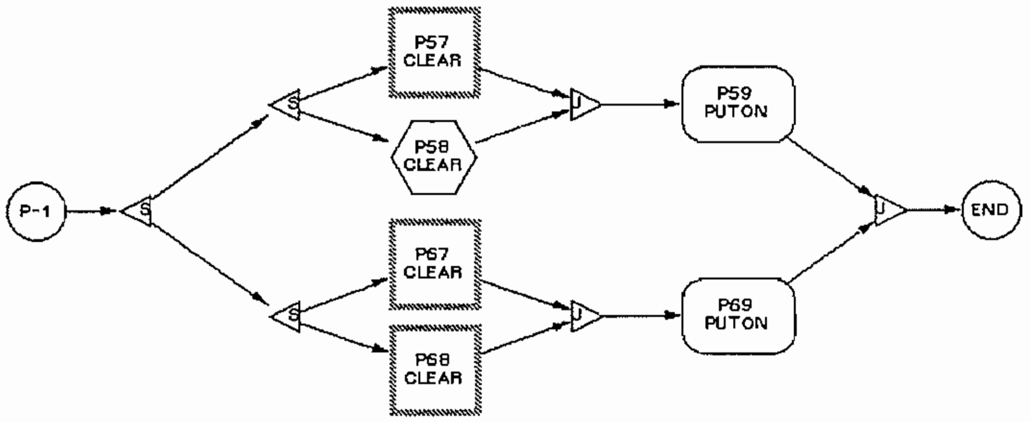


Figure 16: A SIPE-2 Plan drawn with Custom Layout Algorithm.

In Figure 15, all purely vertical edges are plan-critic edges, while the remaining edges are operator edges (ignoring the two line segments cutting across the upper right, which are parts of plan-critic edges that are off the screen). One layout parameter allows the successors of a split node to be staggered horizontally by a given fraction of the node width when there are more than two successors. In Figure 15, the successors of splits are staggered horizontally by four-tenths of a node width. Staggering makes the drawing easier to understand since the three leftmost vertical edges would be coincident without staggering. The Grasper-CL layout algorithms do not produce aesthetic drawings because the Grasper graph does not distinguish between operator and plan-critic edges. Even if such a distinction were added, Grasper-CL still has no layout algorithm designed specifically for context-free graphs.

The power provided by Grasper-CL is evident when comparing the SIPE-2 GUI to the earlier SIPE GUI. In SIPE, only plans could be drawn, there was no highlighting of nodes to visualize relationships within a plan, and there was little flexibility in customizing the final drawing. The user could set some parameters for a drawing, primarily the size of nodes and the label of each node, but could not change the appearance of the drawing in any other way. Nodes could not be moved or displayed with different icons.

SIPE-2, with its Grasper-based GUI, allows a wide range of customizations to be made easily to its drawings by users. Example capabilities include moving nodes to new locations (either individually or in groups), changing the icons of nodes (for example, from a rectangle to a diamond), changing icons to surround the label instead of being a fixed size, and rescaling the drawing either along the x-axis or the y-axis. Because of the ease of constructing and laying out drawings in Grasper-CL, SIPE-2 can draw graphs depicting any important data structure in the system. SIPE-2 also supports several options for visualizing relationships within a plan.

11.1.4 Viewing the Execution of SIPE-2 Algorithms

SIPE-2 algorithms can take several minutes or even hours to run. In particular, generating a large plan or monitoring its execution can be a lengthy process. It is often desirable for the user to get visual feedback about what the system is doing while these algorithms are running. Although SIPE had no such capabilities, SIPE-2 provides them easily by making use of Grasper-CL.

SIPE-2 has an interactive planning mode that gives the user control over certain planning decisions. During interactive planning, the GUI flashes each node in a high-level plan as it is fleshed out at a lower level of abstraction, and highlights any node about which the user is making a decision. Thus, if the user is asked to assist SIPE-2 with a planning decision, they see exactly what part of a nascent plan the question refers to. This capability is invaluable for large planning problems. The GUI can also be used during the execution of a plan to highlight nodes during their execution.

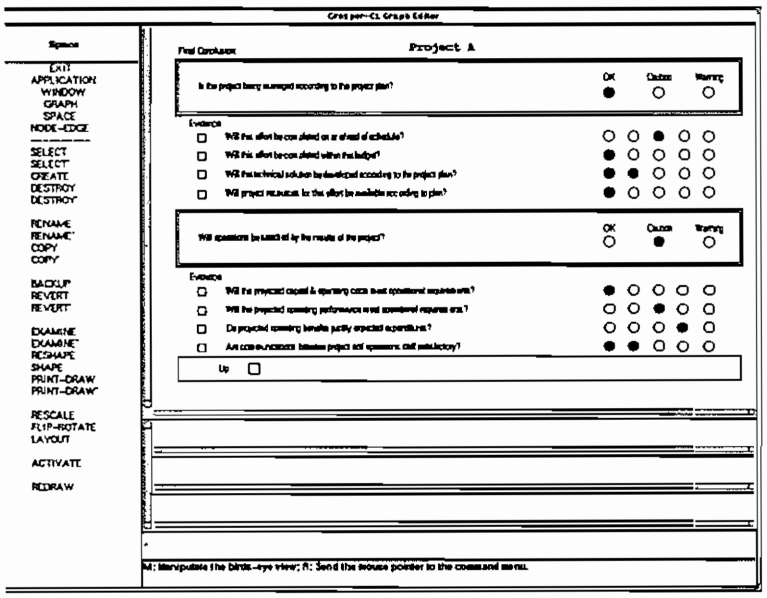
The interactive planner also incrementally highlights and draws parallel links between actions as they are added during planning. In particular, it flashes the first node, then draws the ordering link, and then flashes the second node. This gives an excellent visual depiction of plan growth.

Another useful option allows the user to suspend a SIPE-2 algorithm in order to use the GUI to draw or manipulate graphs, and then continue the algorithm. For example, suppose the user can direct the choice of a planning operator during the generation of a plan. The planning can be suspended, the GUI can be used to draw the alternative operators, and planning can be continued once the choice is made.

11.2 Other Applications

Grasper-CL is in use as the GUI for a number of other AI applications. It creates, edits, and displays probabilistic data flowgraphs in the GISTER evidential reasoning system [8]. It displays schedule graphs for the space telescope as part of a scheduler under development at NASA [15]. It displays inference channels within a graph that represents a database schema as part of a database security project at SRI [2]. It creates, edits, and displays expert procedures as flowcharts in the PRS^{TM} real-time expert system.

The space of node and edge shapes defined by the Grasper-CL shape parameters is so large, and the concept of a graph is so general, that the Grasper-CL system can transcend the role of a traditional grapher to become a general graphical-interface construction kit. For example, Grasper-CL was used in an expert system to construct an electronic multiple-choice questionnaire that accepts a problem definition from the user, as shown in Figure 17. Each of the mutually exclusive multiple-choice boxes (radio buttons) is a Grasper-CL node of fixed size whose node label is invisible. Each phrase is a Grasper-CL node whose node icon is invisible, and whose label is the phrase. The action associated with each node yields radio-button behavior. Multiple Grasper-CL spaces are used to store multiple "pages" of



Final Conclusion

Project A

Is the project being managed according to the project plan? OK ● Caution ○ Warning ○

Evidence

- ☐ Will the effort be completed on or ahead of schedule? ○ ○ ● ○ ○
- ☐ Will the effort be completed within the budget? ● ○ ○ ○ ○
- ☐ Will the technical solution be developed according to the project plan? ● ● ○ ○ ○
- ☐ Will project resources for the effort be available according to plan? ● ○ ○ ○ ○

Will sponsors be satisfied by the results of the project? OK ○ Caution ● Warning ○

Evidence

- ☐ Will the projected capital & operating costs meet operational requirements? ● ○ ○ ○ ○
- ☐ Will the projected operating performance meet operational requirements? ○ ○ ● ○ ○
- ☐ Do projected operating benefits justify expected expenditures? ○ ○ ○ ● ○
- ☐ Are communications between project and sponsors staff satisfactory? ● ● ○ ○ ○

Up ☐

M: Manipulate the birds-eye view; R: Send the mouse pointer to the command menu.

Figure 17: A multiple-choice questionnaire GUI implemented using Grasper-CL.

a long questionnaire. Other Grasper-CL applications use node and edge shapes that yield graphical gauges. The advantage of using the Grasper-CL system for tasks other than displaying traditional graphs is that Grasper-CL constitutes a portable, high-level tool for creating user interface in which the interfaces itself becomes a declarative data structure that can be saved persistently and manipulated with tools such as the interactive graph editor.

12 Related work

A number of other graph display systems have been written in Lisp. Most are difficult to analyze in detail because very little has been published about them. Most are described in only a few pages in a user manual that provides little information on their internal operations. The comments in Section 2 apply to virtually all of these graphers.

The capabilities of the different graphers are all about the same. For example, they all support the same classes of graph-based user interfaces, and they all use the same simple tree-based graph layout algorithm. Because the authors of these systems have not published detailed descriptions of them, there may be a tendency for new authors to reinvent rather than improve upon past capabilities. Also, the authors of these other graphers appear to have largely ignored research by computer scientists outside the Lisp community.

Xerox Corporation implemented a grapher as part of Interlisp that ran on the D-machine series of computers. Researchers at the USC Information Sciences Institute developed the ISI Grapher [13] and the SIMS Grapher. BBN researchers have implemented a Common

Lisp grapher[14], and the CLIM window system includes a grapher.

Notable capabilities of these systems include the following. The ISI Grapher, the SIMS Grapher, and the BBN grapher all include a birds-eye view display. The SIMS Grapher includes an incremental graph browsing mode like that described in Section 10.

None of these graphers have an open architecture like that of Grasper-CL (see Figure 1). None have a graph abstract datatype that is fully accessible to the programmer and that allows graphs to be saved to disk. None have an assortment of graph layout algorithms that can be composed to create hierarchical layouts. None have the wide variety of shape parameters nor shape-parameter defaults that can be overridden. None provide the interactive display module that simplifies the construction of highly interactive graph-based applications.

13 Acknowledgments

Some aspects of the incremental subgraph browsing capabilities described in Section 10.1 were inspired by the capabilities of the ISI SIMS Grapher. This work was supported by the DARPA/Rome Laboratory Planning Initiative under Contracts F30602-91-C-0039 and F30602-90-C-0086.

References

[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] T. D. Garvey, T. F. Lunt, X. Qian, and M. E. Stickel. Toward a tool to detect and eliminate inference problems in the design of multilevel databases. In *Proceedings of the Sixth IFIP WG 11.3 Workshop on Database Security*, August 1992.

[3] T.R. Henry. *Interactive Graph Layout: The exploration of large graphs.* PhD thesis, University of Arizona, 1992.

[4] IntelliCorp. *KEEworlds Reference Manual*, 1986.

[5] Peter D. Karp, John D. Lowrance, and Thomas M. Strat. *The Grasper-CL Documentation Set.* Artificial Intelligence Center, SRI International, Menlo Park, CA, June 1992.

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

[7] T. Koschmann. Designing a browser to support multimethods and method combination. *LISP and Symbolic Computation*, 4(2):143–154, 1992.

[8] John D. Lowrance, Thomas D. Garvey, and Thomas M. Strat. A framework for evidential reasoning systems. In *Uncertain Reasoning*, pages 611–618. Morgan Kaufmann Publishers, 1990.

[9] E.B. Messinger. *Automatic Layout of Large Directed Graphs.* PhD thesis, University of Washington, 1988.

[10] S. Moen. Drawing dynamic trees. *IEEE Software*, pages 21–28, July 1990.

[11] R. Neches. Acquisition of knowledge for sharing and reuse. In *Proceedings of the Knowledge Acquisition Workshop*, October 1991.

[12] E. Rich. *Artificial Intelligence.* McGraw-Hill, New York, NY, 1983.

[13] G. Robbins. The ISI Grapher: A portable tool for displaying graphs pictorially. In *Proceedings of Symboliikka '87*, Helsinki, Finland, August 1987.

[14] J. Sussman. The grapher. Technical Report BBN TR 6876, BBN Laboratories Inc., Cambridge, MA, July 1988.

[15] K. Swanson, M. Drummond, and J. Bresina. An application of artificial intelligence to automatic telescopes. Technical Report FIA-92-26, NASA Ames Research Center, 1992.

[16] D. E. Wilkins and R.V. Desimone. Applying an AI planner to military operations planning. In M. Fox and M. Zweben, editors, *Intelligent Scheduling.* Morgan Kaufmann Publishers, 1992. in press.

[17] David E. Wilkins. *Using the SIPE Planning System: A Manual.* SRI International Artificial Intelligence Center, 333 Ravenswood Ave, Menlo Park, CA, 1992.

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