3-D Visualization of Program Information (Extended Abstract and System Demonstration)

Steven P. Reiss[1]

Department of Computer Science
Brown University
Providence, RI 02912
(401)-863-7641, spr@cs.brown.edu

Abstract

This paper details our work on providing 3-D visualization of program information. We have a project currently underway that attempts to use 3-D workstations to provide insight into programs, their structure and their execution, through the use of a variety of user-definable displays. We provide a variety of different presentation styles and utilize a variety of different layout methods and heuristics.

1 Introduction

This paper describes the visualization aspects of a system for viewing abstract data, particularly information about programs. Our eventual goal is to provide a system where the programmer can specify what information should be displayed and how it should be displayed with a minimal amount of work and where the displays provide the maximum amount of information in an intuitive context[19]. We are designing a framework to accomplish this. This framework is based on a package we have written for abstract 3-D visualization, PLUM. In addition, we have implemented a package for hierarchically browsing over the data, PEACH, and a package, TWIG, for mapping from arbitrary data structures representing program information into the graphical representation structures required by PLUM. We are currently working on a visual language for specifying both the information to display and the visual representation of this information.

1. Support for this research was provided by the NSF under grants CCR9111507 and CCR9113226, by DARPA order 8225, by ONR grant N00014-91-J-4052, and by support from Sun Microsystems.

2 Background

While there has been substantial work on program visualization, most of this work has been directed toward providing specific visualizations such as a call graph or a class browser, and little has been directed toward a generic framework. The work that is closest to our approach includes our earlier efforts on data and program visualization [15-18], work related to the display of user data structures [1,9,13], work directed at graphical editing [5,11,14], work on systems for algorithm animation [2-4,22], and visualization efforts that attempt to use a single paradigm for a variety of applications [7,8,10,12,20,23].

3 Moving From 2-D to 3-D

With the exception of some of the more recent work on algorithm animation and some simple experiments, almost all of the work on program visualization has involved 2-D presentations. We wanted to determine whether the expense and complexity of using three dimensions could significantly improve program visualization and affect program understanding. To do this, however, we needed to determine how to use the extra dimension.

Most of the solutions that we and others have devised for utilizing 3-D for data visualization involve extending what is normally a 2-D representation into a 3-D one. This is desirable since it maintains a 2-D philosophy and presentation, allowing the viewer to see all the data at once while also allowing the additional dimension to be used for a variety of purposes. There are a variety of techniques that can be used here.

Several of these techniques do a 2-D layout and then extend the graph into the third dimension using some property of the data. Other methods take the 2-D layout and provide a 3-D organization of the information. Other solutions to moving from 2-D to 3-D space attempt to actually use the full capabilities of three dimensions without attempting to preserve a full 2-D view from some perspective. More complex examples include using the third dimension to represent time.

Another visualization technique that makes effective use of three dimensions is to provide several different 2-D visualizations simultaneously. One of the presentation methods we provide, for example, allows the user to select one hierarchy to be displayed in the XY plane, one hierarchy to be displayed in the YZ plane, and a third hierarchy to be displayed in the XZ plane. We place nodes so that each of the hierarchies can be seen if viewed directly on and so that the user can see connections between the different hierarchies by viewing the result from different perspectives. A similar strategy has been proposed and used by others.

4 PLUM Structure

In order to experiment with 3-D visualizations, we needed to develop a framework that allows different presentation styles. The framework we chose is object-oriented, using different flavors of objects to represent the different presentation methods. This framework is also hierarchical so that different visualizations can be easily combined. For example, the time-based views of a dynamic call graph takes as subobjects the original call graph nodes representing the called functions, and arcs representing the actual calls. It operates by determining the position of each of the dynamic call nodes based on the position of the corresponding node in the original call graph and the entry and exit times. Using an object-oriented approach also makes extensibility easy. New classes of objects can be defined to reflect new presentation styles. Moreover, a presentation style can be specialized by subclassing the object that represents it.

PLUM presentation objects are characterized by a flavor that denotes the type of object. Each object has an associated set of properties that parameterize the presentation. These are described in section 6.0.

Objects can have both components and constraints. Components are used to describe other objects that are children of the given object. Constraints are containers for additional information that is to be associated with an object.

PLUM computes a layout in three phases. It assumes that the application has set up a tree of objects, components, and constraints. The first phase computes the desired size of each of the objects. This is the size, determined by the object itself, that it would ideally like to be drawn. This is done bottom up, with each node of the tree first asking its component nodes to determine their size and then using the resultant information to determine the size of the node itself. The next phase involves layout. This is generally done top-down. Each object is responsible for determining the actual size and position of each of its components. The actual size generally will correspond to the desired size that the component specified. However, the size can also be larger or smaller depending on the needs of the parent object. Positions are defined relative to the center of the parent object. Once the object determines the size and position of its components, it has each of the components compute the size and position of their components. Having this pass be top-down allows an object to know its actual size before it has to lay out its components. The third phase involves actually drawing the components. This is typically done top-down as well since the parent object provides background for the children.

5 Graphical Presentation Objects

The basic objects offered by PLUM can be divided into three categories. The first category defines basic objects. These are object that have a concrete screen representation. These include data objects, arc objects, and light objects.

The second category of presentation objects are those that provide layout services, i.e. placement and sizing of their component objects. These include tiled objects, layout objects, sized layout objects, tagged objects, time sequences, and 3-D trees. Tiled objects represent 3-D rectilinear tilings that are solved using a system of constraints. Layout objects and sized layout objects represent arbitrary layouts of nodes and arcs that are positioned using the layout heuristics described in section 7.0. Tagged objects allow a tag to be attached to a layout or other object. Time sequences use the Z dimension to represent time, for example to show the dynamic call graph for a program. 3-D trees are similar to Xerox's cone trees.

The third category of presentation styles includes styles that control both layout and presentation. These include scatter plots, file objects, and plot layout objects. File objects are similar to Bell Laboratories SeeSoft displays. Plot layout objects allow the application to specify the size or position of layout components and position any unspecified nodes using a relaxation algorithm.

6 Styles in PLUM

Properties serve a variety of functions in specifying the various types of presentation objects. They control the graphical presentation of the objects, specify parameters that control the layout, and provide data to be used in the presentation. As such, it was important that PLUM provide a powerful and convenient mechanism for specifying properties.

The basic notion is based on styles. A style is a collection of properties each of which is an attribute-value pair. Properties are grouped into styles as a matter of convenience. It is often the case that a set of different objects will share a common set of properties. Styles allow this set to be specified once and then simply associated with each of the objects. Styles are designed to support standard definitions like this while still allowing properties to be overridden for individual objects. They also are designed so that styles from difference sources can be easily merged. This allows a style that is specified for selection to be merged with the default style for an object. Finally, styles are designed to allow the setting of default properties for each flavor of presentation object and for drawing.

Styles are implemented as objects where values are determined by object-oriented delegation rather than inheritance.

Objects have several styles associated with them. In addition to a style computed by the system that represents the current selections, they have a base style that is generally defined by the object to specify default values for object-specific properties, a user style for normal settings, an override style for priority settings, and a child style to indicate settings for their subobjects.

7 Layout Methods in PLUM

Must of the work done by the presentation objects involves layout, i.e. placement of subobjects. A variety of different layout strategies are evident, some attempting to use layout to convey information, e.g. using depth to indicate the amount of time spent in a routine or using Z to represent time. Others just attempt to make the layout look "good" according to some abstract criteria.

The simplest layout method is used for layout objects and sized layout objects. These methods allow arbitrary nodes and arcs and simply attempt to do graph layout in 3-D, typically while presenting a 2-D view from the front of the display. Graph layout has been extensively studied in two dimensions [6]. The problem is one of placing nodes and arcs to produce an aesthetically pleasing graph. This is generally translated into more specific problems such as reducing arc length and the number of crossings or of emphasizing symmetry. While we provide a variety of approaches in our 2-D layout packages, the algorithm of choice for program data has been one based on level graphs [21] since it tends to emphasize hierarchy and since it generally produces a reasonable looking result.

Moving graph layout algorithms from two to three dimensions is not trivial. The first problem is determining what "looks good" in three dimensions. Because 3-D graphics imply that the user is going to move around and look at the graph from different perspectives, assumptions based on the user's view may not be valid. For example, the heuristic of minimizing crossings is meaningless. Given any two arcs in three space that do not intersect, we can find a perspective where they do not cross and a second perspective where they do cross. Since most arcs will not physically intersect in three space, the number of crossings will vary with the perspective.

A second problem is that 3-space offers many more degrees of freedom. In two dimensions there are two alternatives to laying out a level graph, representing the levels as either rows or columns, and the resulting graphs are identical except for orientation. In three dimensions one has three alternatives for how to represent levels. Moreover, once the leveling is done, each level can be potentially represented by a plane and hence by an arbitrary 2-D layout. One could, for example, apply a 2-D level graph algorithm to the remaining nodes, i.e. do leveling twice. Alternatively, the algorithm could place the nodes in a circle as in cone trees.

A third problem that arises is that we want to use the third dimension to convey information and not just to provide more space for layout. This means that we have to find layout methods that reflect properties of the underlying objects. For example, layout methods must be able to assign a Z coordinate to a node based on its accumulated run time or what file its in or how distant it is from a set of selected nodes that the user is focusing on.

In PLUM we have implemented a flexible approach to 3-D layout to experiment with different algorithms and to gain experience with what works and what does not. Our approach allows layout methods to work in various ways. Some methods, such as leveling, work for one dimension and depend on another layout method to handle the remaining dimensions. Other layout methods are comprehensive, working in all three dimensions at once. Still others, such as local optimization, don't compute the layout in any dimension, but instead modify a layout that is already present. All the layout methods allow values to be defined by the application or by the user. Each coordinate can be given a default relative or a default absolute value. Relative values identify the location in the array that is used by layout objects. These are typically used to represent program assigned values. Absolute values can be used to exactly reflect user manipulations of the underlying objects. All the layout algorithms are also parameterized using properties.

The layout methods that handle only one dimension include level graphs, level rankings, unique values, and application-specified values.

Other layout methods handle all remaining dimensions. These can be used as a top-level method or as a secondary method to some of the above to fill in the remaining values. They include depth-first layout, breadth-first layout, averaged layout, and orthogonal layout. The latter attempts to display multiple hierarchies in the XY, XZ, and YZ planes.

Finally, we currently provide one post-processing optimization that uses a relaxation algorithm.

8 Animation in PLUM

One of the key features provided by PLUM is automatic animation. Animation is necessary for 3-D visualization. PLUM provides animation in two ways. The simpler is to allow the user to move the camera position so as to fly through the object. The more complex allows arbitrary changes to the presentation objects to be made.

PLUM is designed to be used by an application in an edit-display cycle. The application first sets up a top level presentation object and then asks that it be displayed. Then, in response to a user request or a program action, it either edits the current presentation objects or creates a new top level presentation object. Editing can involve setting new property values for the presentation objects, creating new presentation objects and attaching them as components of existing objects, removing object components, adding or modifying constraints, or selecting or deselecting objects. Once a series of edits is complete or a new top level object has been defined, the application informs PLUM that the edit is complete. Unless told otherwise, PLUM will attempt to animate the transition from the current display to the display of the modified objects.

The first stage in this automatic animation process involves identifying what has changed. The second phase occurs at the start of updating the display. Here PLUM attempts to match old and new objects and to save the old display settings for matched objects. Once the old and new objects have been compared, PLUM computes the new

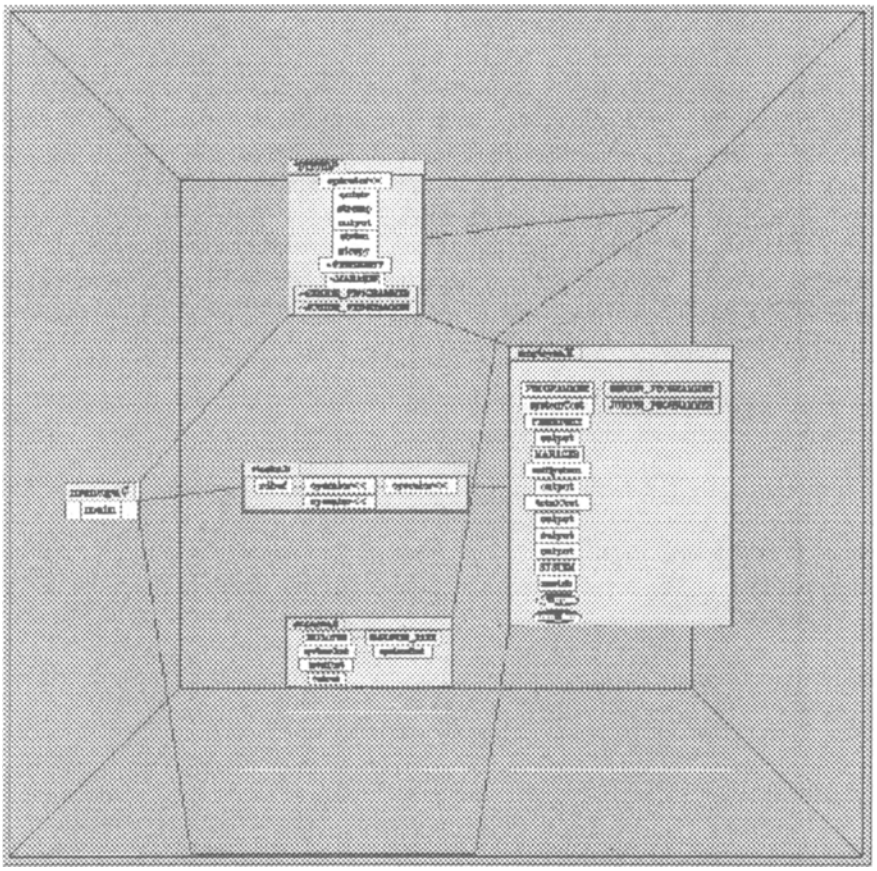

FIGURE 1. 3-D call graph display

presentation. The new presentation is then drawn by making passes over the resultant object structure to compute the display list corresponding to each object. The object properties used in each pass are automatically determined by PLUM using interpolation between the original and desired values.

9 Experience with PLUM

We have been working on PLUM and the related packages for abstraction visualization for about two years. During that time we have rewritten most of PLUM at least once in attempting to find the proper abstractions and interface. The current system comprises about 25,000 lines of C++ code. Examples of the system are shown in the figures. Figure 1 shows an example of a 3-D call graph. Figure 2 shows a call graph where tagged objects are used to represent the files. Figure 3 shows a top-down view of a dynamic call graph display using a time sequence object. An example of a tree

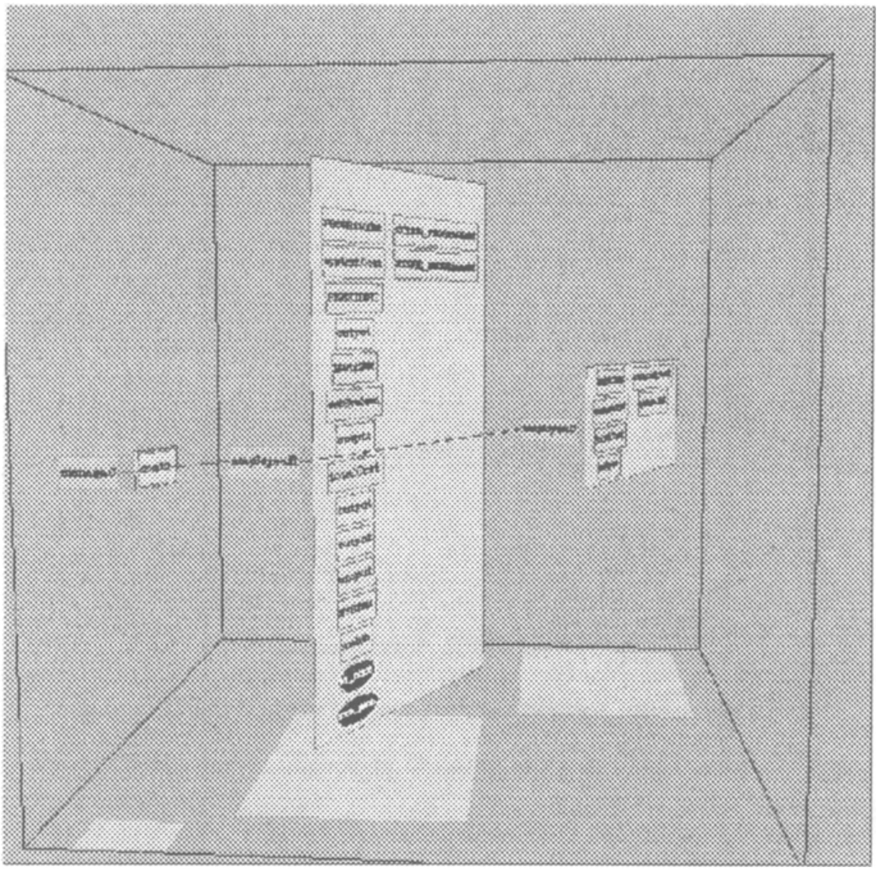

FIGURE 2. Call graph using tagged objects

layout is shown in figure 4. Figure 5 shows file objects being used in a call graph display.

Much of our experience with PLUM has been positive. The framework provided by PLUM makes it easy to add new presentation styles. This is due to the use of hierarchy to simplify what each presentation has to do, the general notions of properties, components and constraints that are supported by the system. We have been able to integrate a variety of different presentations into this framework in a natural way. Adding a new presentation style can generally be done in a day or less, but additional time is often required to fine tune the graphical presentation.

The interface between PLUM and the application also seems to be the right one. The application defines objects, components, and constraints. Styles and properties are associated with objects. Properties can be set for components and constraints. Because

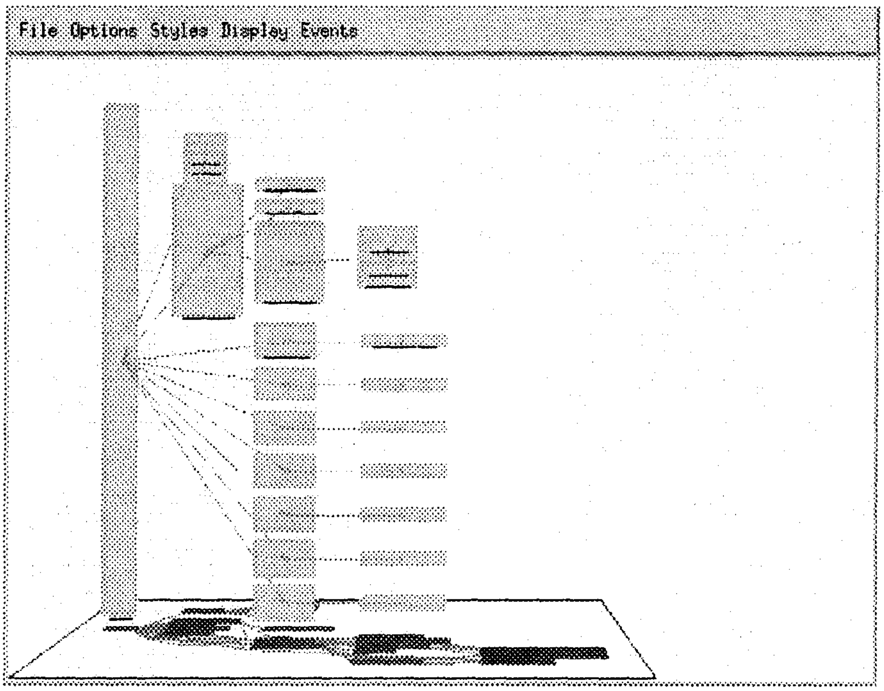

FIGURE 3. Time sequence objects in dynamic call graph display

object, components, constraints, and styles are all generic, the size of the interface that is required is quite small given the complexity of the system. The use of automatic animation allows the application to compute the new presentation without having to specify how it differs from the old. This often greatly simplifies the application.

The weak points of PLUM generally relate to performance. The presentation is too slow for convenient viewing and interaction, especially for displaying large, complex structures. There are three reasons for this. The primary one seems to be the performance of the underlying graphics system independent of PLUM. A complex presentation can take seconds to redraw even if no work is done other than changing the camera position and retraversing the display list. This means that animations are very choppy and the user does not get the sense of 3-D that is necessary for understanding the display.

The second performance bottleneck lies in the system structure. Our currently implementation of PLUM has PLUM run as a separate process. The application interface to PLUM translates each interface command into a message that is then sent to the PLUM server where it is implemented. Even though the messages are sent using a shared memory buffer, this process is time consuming. A complex display can involve several thousand messages, each of which has to be encoded, placed into the memory buffer, and then decoded. While it is convenient to architect the system as a separate

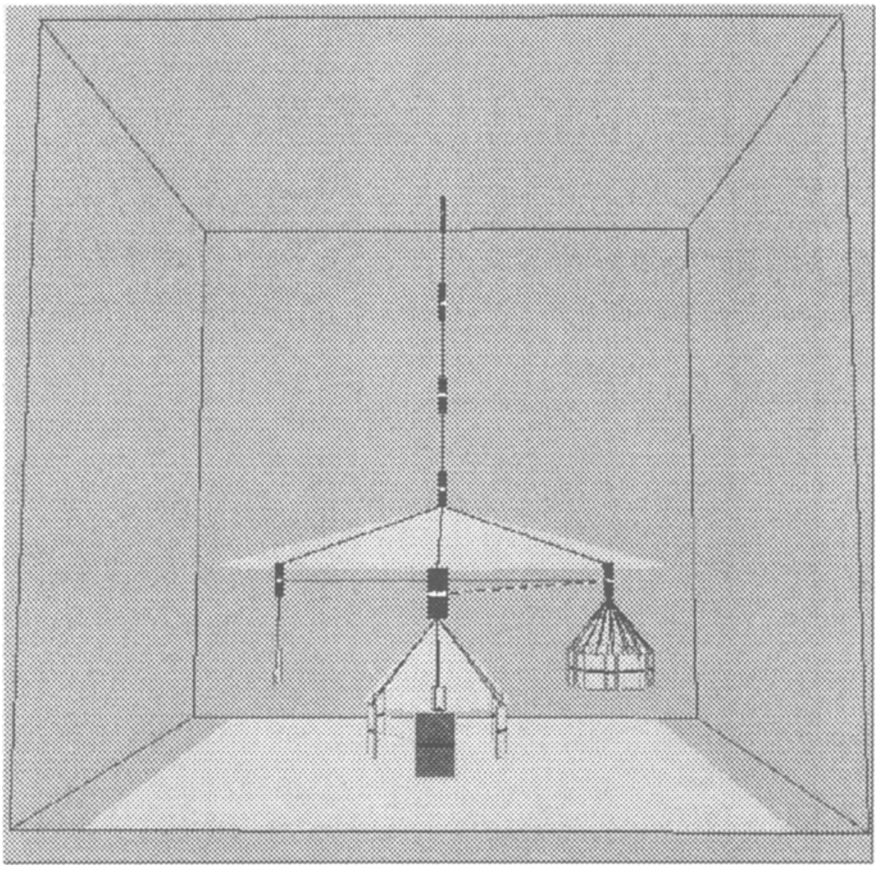

FIGURE 4. Tree layout display of call graph

process for debugging and to simplify the applications, the current structure will allow us to incorporate PLUM into the application if this ever becomes the primary bottleneck.

The third performance problem is more difficult to deal with. This involves the implicit architecture of a system using PLUM where PLUM creates shadow objects for each application object. The use of such indirection and the cost of maintaining shadows and transferring and maintaining the properties of the shadows as the original objects change constitutes a substantial portion of what an application using PLUM needs to do. This is aggravated by the browser that we currently have that sits in front of PLUM. This browser creates its own shadow objects and hence a second level of indirection between the user object and the drawing object. One of the eventual goals we have for program visualization would be to look at trace data where there can be a very large number of objects that need to be considered in computing the display. Creating multiple shadow copies of each of these objects seems impractical.

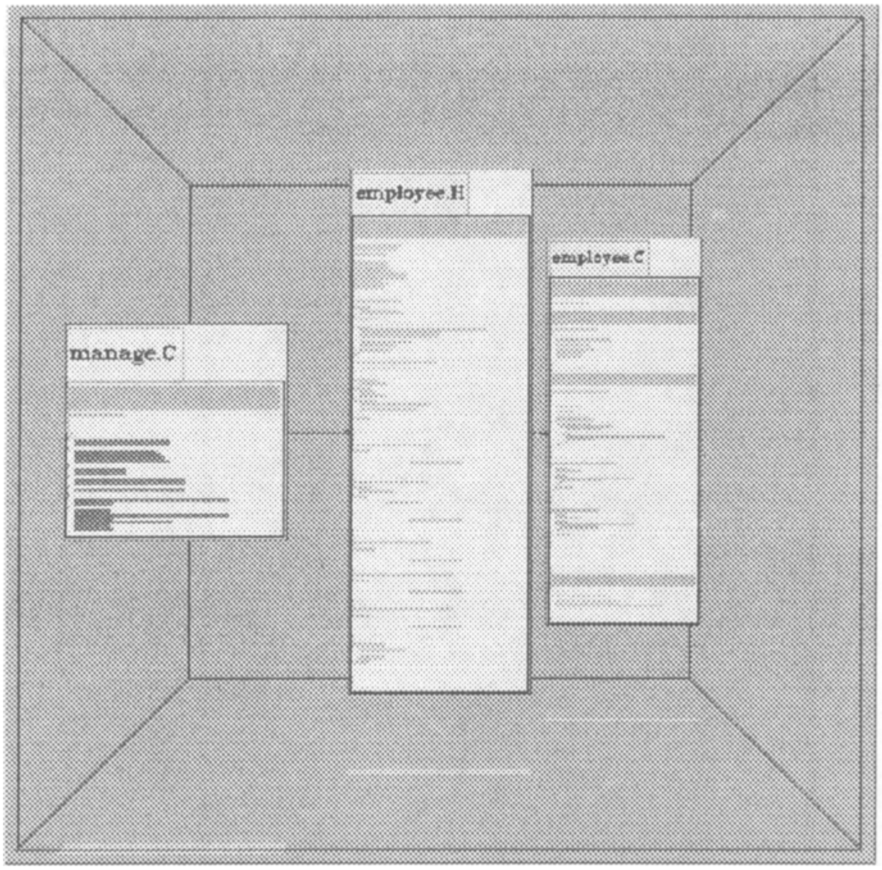

FIGURE 5. File object display showing function properties

10 References

1. David B. Baskerville, "Graphic presentation of data structures in the DBX debugger," UC Berkeley UCB/CSD 86/260 (1985).

2. Marc H. Brown and Robert Sedgewick, "Techniques for algorithm animation," *IEEE Software* Vol. **2**(1) pp. 28-39 (1985).

3. Marc H. Brown and John Hershberger, "Color and sound in algorithm animation," *Computer* Vol. **25**(12) pp. 52-63 (December 1992).

4. Marc H. Brown and Marc A. Nojork, "Algorithm animation using 3D interactive graphics," DEC Systems Research Center (1992).

5. Jacques Davy, "GoPATH programmer's guide," Bull Imaging and Office Solutions (December 1992).

6. P. Eades and R. Tamassia, "Algorithms for automatic graph drawing: an annotated bibliography," *Networks*, (1993).

7. Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr., "Seesoft -- a tool of visualizing software," AT&T Bell Laboratories (1991).

8. Belinda B. Flynn and David Maier, "Specification and generation of displays for complex database objects," Oregon Graduate Institute of Science and Technology (1992).

9. Sadahiro Isoda, Takao Shimonmura, and Yuji Ono, "VIPS: A visual debugger," *IEEE Software* Vol. 4(3) pp. 8-19 (May 1987).

10. Clinton Lewis Jeffrey, "A framework for monitoring program execution," U. Arizona Technical Report TR 93-21 (July 1993).

11. Mark A. Linton and John M. Vlissides, "Unidraw: A framework for building domain-specific graphical editors," *Proc. UIST '89*, pp. 158-167 (November 1989).

12. Jock D. Mackinlay, George G. Robertson, and Stuart K. Card, "The perspective wall: Detail and context smoothly integrated," *Proc. CHI'91*, pp. 173-179 (April 1991).

13. Brad A Myers, "Incense: a system for displaying data structures," *Computer Graphics* Vol. 17(3) pp. 115-125 (July 1983).

14. Brad A. Myers, Dario A. Guise, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal, "Garnet: Comprehensive support for graphical, highly interactive user interfaces," *IEEE Computer*, pp. 71-85 (November 1990).

15. Steven P. Reiss and Joseph N. Pato, "Displaying program and data structures," *Proc 20th Hawaii Intl Conf System Sciences*, (January 1987).

16. Steven P. Reiss, "Working in the Garden environment for conceptual programming," *IEEE Software* Vol. 4(6) pp. 16-27 (November 1987).

17. Steven P. Reiss, Scott Meyers, and Carolyn Duby, "Using GELO to visualize software systems," *Proc. UIST '89*, pp. 149-157 (November 1989).

18. Steven P. Reiss, "Connecting tools using message passing in the FIELD environment," *IEEE Software* Vol. 7(4) pp. 57-67 (July 1990).

19. Steven P. Reiss and Manojit Sarkar, "Generating program abstractions using an object-oriented database," Brown University Department of Computer Science (1992).

20. George G. Robertson, Jock D. Mackinlay, and Stuart K. Card, "Cone trees: Animated 3D visualizations of hierarchical information," *Proc. CHI'91*, pp. 189-194 (April 1991).

21. L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan, "A browser for directed graphs," *Software Practice and Experience* Vol. 17(1) pp. 61-76 (1987).

22. John T. Stasko, "TANGO: A framework and system for algorithm animation," *IEEE Computer* Vol. 23(9) pp. 27-39 (September 1990).

23. James Wen, "A three dimensional browser for visualizing orthogonal hierarchies," Brown University (1992).