Visualization for Software Engineering -- Programming Environments

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912
spr@cs.brown.edu
(401)-863-7641, fax (401)-863-7657

Abstract

This chapter describes the visualizations provided by the FIELD programming environment. This system, developed between 1986 and 1992, offered a variety of program visualizations ranging from diagrammatic visualizations of the program structure, to user-definable data structure visualizations, to more abstract visualization of program execution. In this chapter we describe some of the techniques we used to make these visualizations effective and to allow them to be used for large systems. These include a variety of browsing and information encoding techniques that were designed to make maximal use of screen space and to let the user focus on the appropriate information of interest. We also provide our prospective on the effectiveness of these visualizations.

1.0 Overview

The Friendly Integrated Environment for Learning and Development, FIELD, was developed from 1986 through 1992 as an attempt to use workstations effectively for UNIX-based programming. It integrated a wide variety of UNIX tools into a common framework. This framework was enhanced with new tools developed for the environment, both tools for programming support and tools for program visualization.

FIELD integrated the various programming tools using a message-based integration mechanism. This mechanism uses a central message server that the tools (or appropriate tool wrappers) talk to. When the tool starts up, it registers with the message server patterns that describe the messages it is interested in. As it runs, messages are sent to the message server and are redistributed according to these patterns to all interested tools. Messages are typically of two types, commands that request action from another tool, and information that provide data from one tool that might be of potential interest to other tools.

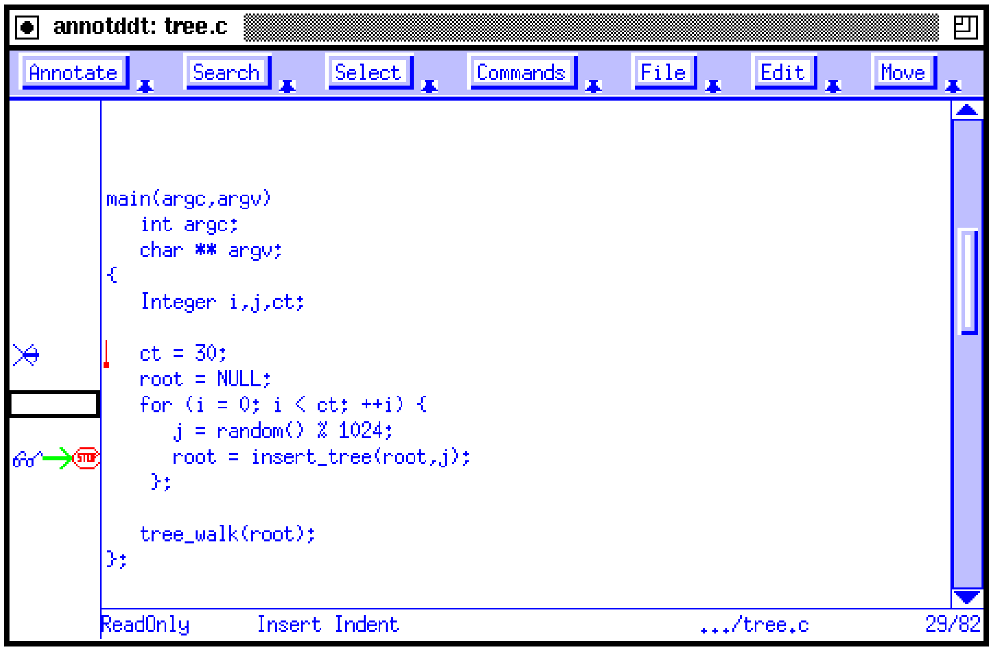
To augment the existing set of UNIX programming tools and to provide a basis for program visualization, FIELD provided several new tools or extended tool wrappers that offered data about a program to other tools through the message server. The primary such tool was the cross-reference database. This tool gathered information about the syntax and semantics of a program and stored it in a relational database for querying by other tools. The information, gathered either by scanning the source or using compiler-generated data, included references, definitions, scopes, calls, include dependencies, function descriptions, class hierarchy links, and class member data. Other tools included a wrapper around both configuration management (*make*) and version control (*rcs*) tools that offered query facilities, and a generic wrapper for the various UNIX profiling tools that again allowed querying by other tools.

In addition to these tools for analyzing static data, FIELD provided a library-based monitoring facility that utilized the message server to provide information about a program's execution while the program was running. This tool worked by inserting its own library between the application and the system libraries to catch calls related to memory allocation or to input/output. It then sent messages describing each allocation or input/output event so that other tools, primarily dynamic visualizations, could display the program in action.

2.0 FIELD's Visualization Tools

In addition to demonstrating how the computation power of workstations could be used to integrate a wide variety of programming tools, FIELD attempted to demonstrate the power and possibilities of program visualization opened up by their graphical capabilities. It offered program visualizations both to provide a better interface to existing and newly developed tools and to offer the programmer insights into program structure and behavior.

The earliest visualization tool provided by FIELD was the annotation editor. The FIELD annotation editor provided a common front end to the program source for both editing a visualization. It was a full-function text editor augmented with an annotation window to the left of the text as can be seen in Figure 1. Annotations were displayed in this window as a descriptive icon. Text associated with an annotation could be displayed at the user's request. Annotations were derived from messages received from the message server so that tools could request an annotation by sending an appropriate message. The editor could be set up to automatically display the line containing the annotation, changing files or position as needed to do so. It could also be set up to display a particular type of annotation by highlighting the corresponding line rather than using an icon. The annotation editor provided



```
main(argc,argv)
   int argc;
   char ** argv;
{
   Integer i,j,ct;

   ct = 30;
   root = NULL;
   for (i = 0; i < ct; ++i) {
      j = random() % 1024;
      root = insert_tree(root,j);
    };

   tree_walk(root);
};
```

```
ReadOnly       Insert Indent                          .../tree.c          29/82
```

FIGURE 1. FIELD annotation editor

both static and dynamic visualizations. Static visualizations included a display of where breakpoints were located in a file and information about errors and warnings from compilation. Dynamic visualization showed the program executing either by a moving icon or by continually highlighting the currently executing line.

The next set of visualization tools offered by FIELD were diagrammatic based on a general purpose structure visualization package originally developed for the GARDEN programming system [GARDEN]. This package, GELO, was designed to as a generic means for displaying 2D structured diagrams. It provided a simple framework based on simple boxes, arcs, rectilinear tilings, and arbitrary layouts. It included constraint satisfaction methods for managing tilings and a variety of placement and routing algorithms for handling layouts. Three FIELD tools were written using GELO directly: the call graph browser *flowview*, the class hierarchy browser *cbrowse*, and the make dependency browser *formview*.

The flowview tool provided a hierarchical view of a program’s call graph as seen in Figure 2. This tool gets information about functions and calls from the cross-reference database. It organizes the information in a hierarchical fashion by grouping functions into files, files into directories, and directories into their parent directory. It provides a variety of browsing techniques aimed at allowing the programmer to focus on particular items of interest. If offers detailed information through a textual information window. It also provides a dynamic view of the program by highlighting nodes as they execute. For example, in Figure 2, the currently executing node is shown in red and the other nodes active on the call stack are shown in green.

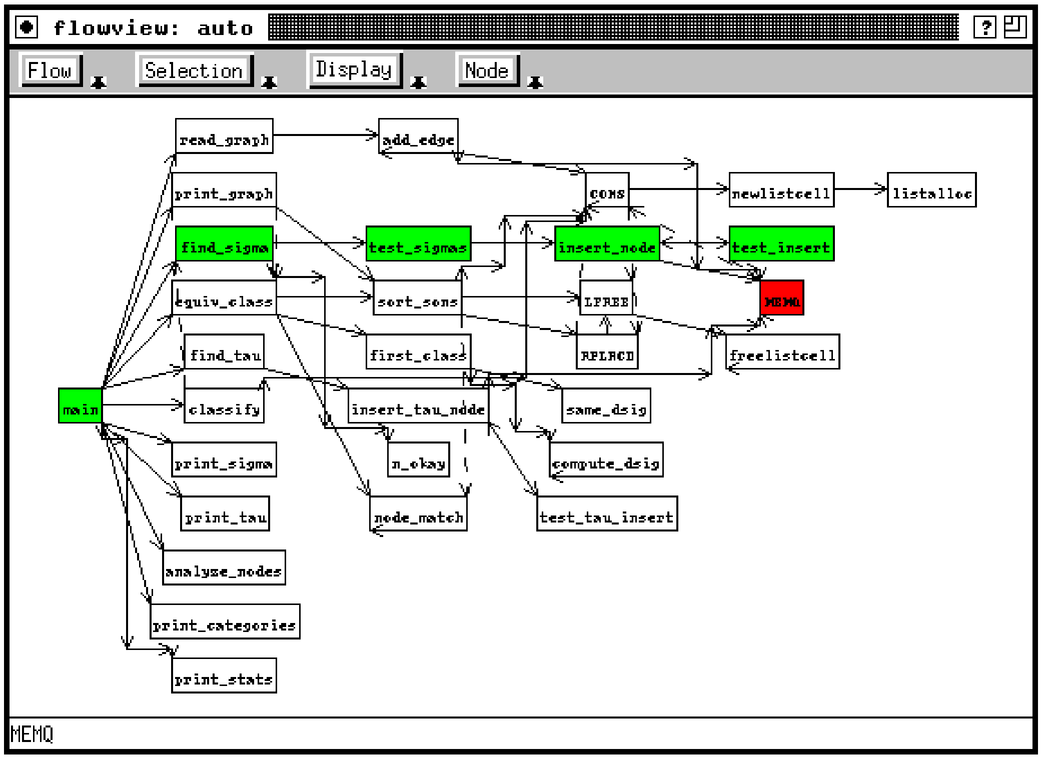


FIGURE 2. Dynamic call graph display

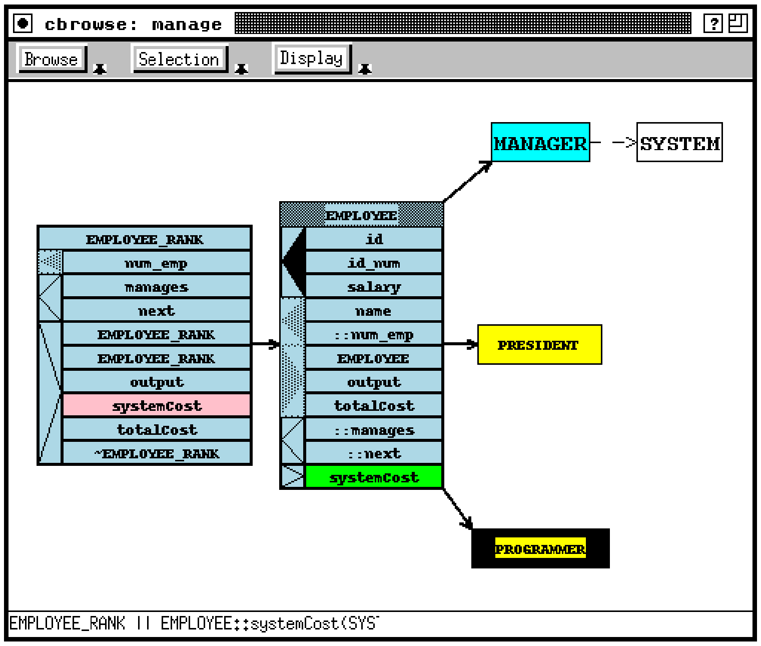


FIGURE 3. Class Hierarchy Display

The second diagrammatic visualization displays the class hierarchy for C++ or Object Pascal programs as shown in Figure 3. This tool also obtains its information from the cross-reference database. It displays information about both the relationships between classes and the methods and fields of each particular class. It

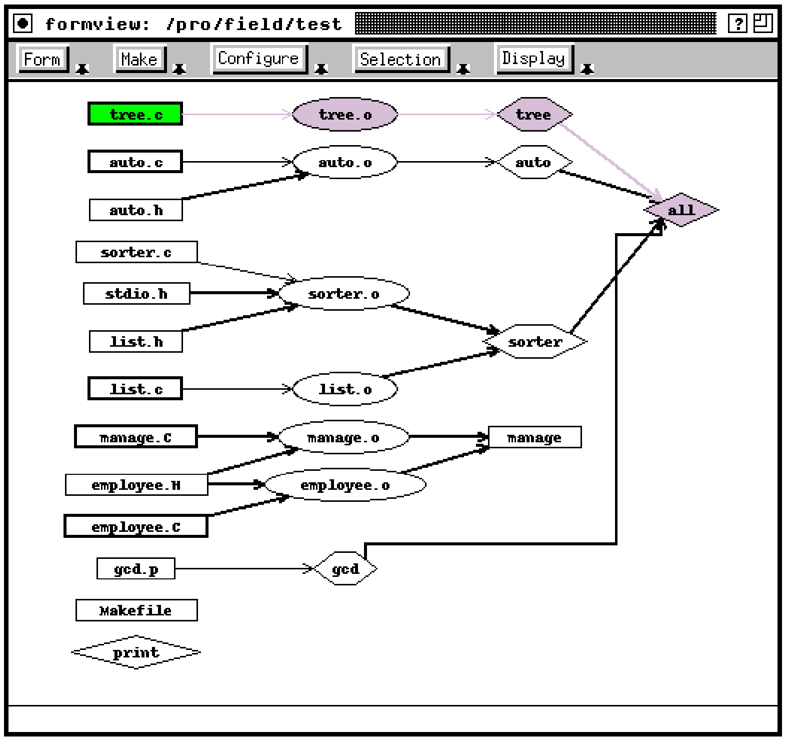
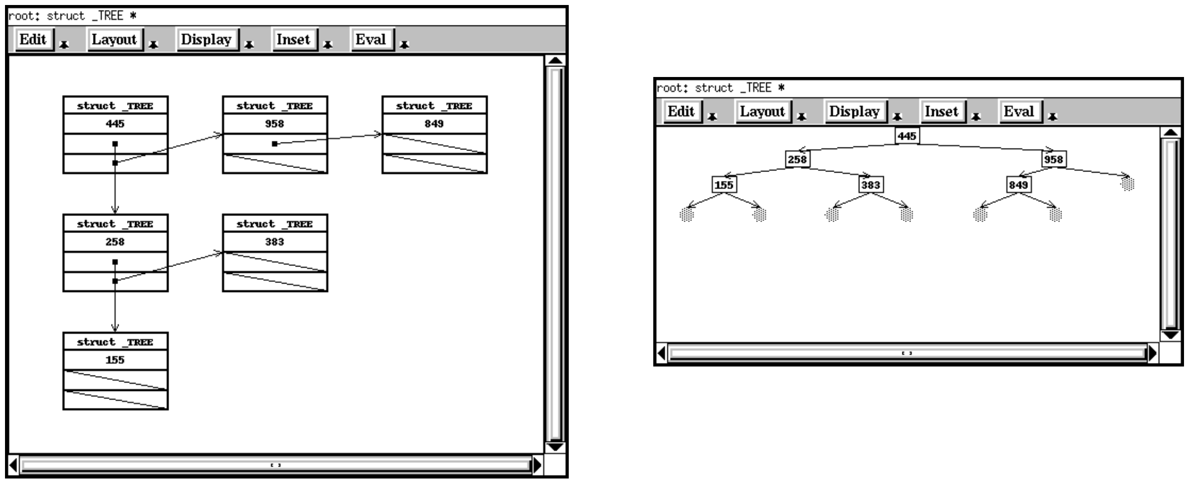


FIGURE 4. Make dependency display

provides a variety of browsing techniques to allow the user to focus on a particular class or member. It uses different visual encodings to illustrate the contents of a class and the different relationships. For example, color is used to indicate the selected member and the relationship of that member in related classes — whether it is inherited, redefined, etc. This tool also provides a textual information window with detailed information about the selected class and dynamic highlighting of methods as they are executed.

The third visualization tool provided by FIELD used the wrapper around *make* and *rcs* to gather information about file and build dependencies and put up a corresponding display. This display, as shown in Figure 4, showed the dependencies and the current state of each file. It uses color to indicate which files are out of date and which targets need to be rebuilt. The type of border around a file box indicates the current check-out state of the file. Different shape nodes indicate different types of targets. The window is interactive in that the user can browse to select targets of interest and can initiate check-in, check-out, and build operations from the display.

A fourth visualization tool in FIELD that uses GELO displays user data structures. This tool comes in two pieces; in addition to the actual display of the data structure, there is a visual editor that allows the user to define how the data should be displayed. For example, Figure 5a shows the default display of a linked tree structure while Figure 5b shows the display after the user has used the visual editor to cause the structure to be displayed as a tree. The data for this display was obtained from the system debugger through the message server. This view of the



a) Default display of a tree b) User-defined display of a tree

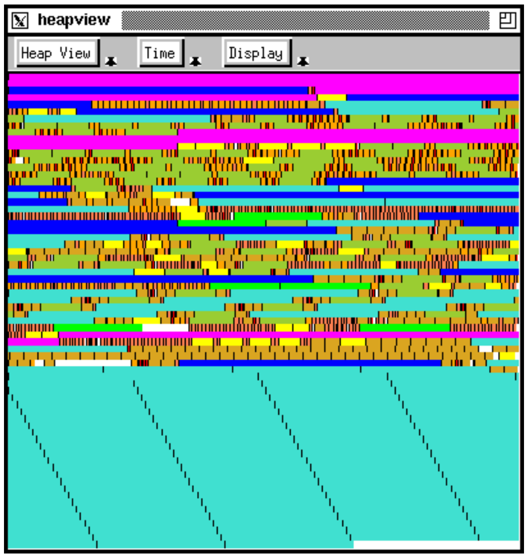
FIGURE 5. Data structure visualization tool

data structure also served as an editor, allowing the user to change the data while the program was being debugged.

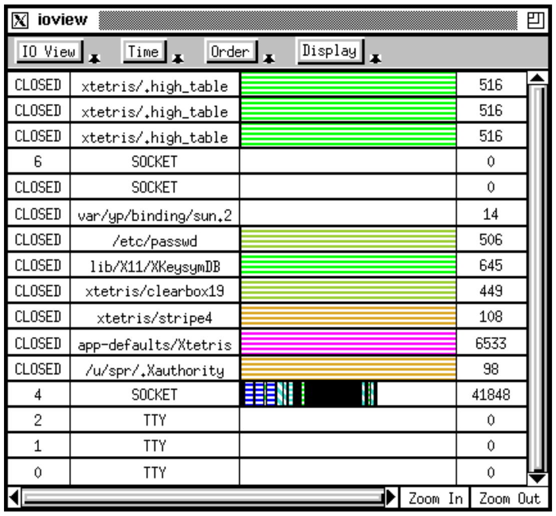
In addition to the diagrammatic visualizations based on the GELO package, we introduced several more abstract visualizations to illustrate program execution. Three such visualizations were developed, a view of the memory allocation and the heap, a view of file input and output, and a graph of various performance statistics. The heap visualizer, shown inFigure 6a, provided a color-coded map of user-allocated memory. Colors could be used to show block size, when the block was allocated, or where the block was allocated from. By clicking on a block, the user could get detailed information about when and where it was allocated. This view has been used to find memory leaks and other anomalous program behavior in a variety of programs. The second tool offered a view of file I/O. In addition to showing the status of all attempted opens, it showed how files were read and written. Colors were used to show either I/O time or block size; fill patterns were used to differentiate reads from writes. An example is shown in Figure 6b.

3.0 Browsing Techniques

While diagrammatic visualizations are common in describing software, they are not particularly good at displaying large amounts of data. The binary for the bulk of the FIELD system, for example, consists of about 7,000 functions and 22,000 calls. It is almost impossible to put up a meaningful display of the resultant graph on a small screen. The effectiveness of the diagrammatic visualizations for moderate-to-large scale software systems then depends on providing effective techniques for providing the “right” display. This means allowing the programmer to easily focus on the part of the overall graph that is of interest to the current situation and providing



a) Heap visualizer



CLOSED	xtetris/.high_table	🟩	516
CLOSED	xtetris/.high_table	🟩	516
CLOSED	xtetris/.high_table	🟩	516
6	SOCKET		0
CLOSED	SOCKET		0
CLOSED	var/yp/binding/sun.2		14
CLOSED	/etc/passwd	🟩	506
CLOSED	lib/X11/XKeysymDB	🟩	645
CLOSED	xtetris/clearbox19	🟩	449
CLOSED	xtetris/stripe4	🟧	108
CLOSED	app-defaults/Xtetris	🟪	6533
CLOSED	/u/spr/.Xauthority	🟧	98
4	SOCKET	▨	41848
2	TTY		0
1	TTY		0
0	TTY		0

b) Input/Output visualizer

FIGURE 6. FIELD visualization tools

as much information as practical within the limits of the display. FIELD's GELO-based browsers attempt to do both. The former is done by providing a range of powerful browsing capabilities. The latter is done through a variety of information encodings.

Browsing in the three diagrammatic visualization tools is based on three techniques: exploiting hierarchy, using names and name patterns, and considering only connected graphs.

Of these techniques, exploiting hierarchy is the most effective, allowing the programmer to quickly focus on the items of current interest. Most large software visualizations can be view hierarchically. The class graph hierarchy was derived directly from the superclass-subclass relationship. The call graph hierarchy was derived by grouping functions into the file they are defined in, files into their directory, and directories up the UNIX directory hierarchy. The dependency hierarchy was determined by the use of recursive invocations of *make*. Where there hierarchies are not unique (as in a class graph where multiple inheritance is used), the tool used depth-first search to identify a hierarchy that was then used. In addition to these natural hierarchies, the call graph browser allowed the user to define new hierarchies on the fly. These hierarchies were defined by name patterns, one pattern to identify candidate names and a second pattern to determine the parent of a given node. These were used to group all methods or to group all methods with the same method name.

Once a hierarchy is established, the various browsers provided the ability to easily collapse and expand nodes. In general three techniques were provided. The mouse could be used to expand or collapse a particular node. This was typically done by having control-Right button collapse a node and selecting an already selected node expand that node. Secondly, one or more dialog boxes were provided where the user could browse over the various nodes being displayed and choose which should be expanded or compacted. The call graph browser presented these options using the hierarchy; the other browsers just presented the nodes directly. Finally, the user could specify as part of the drawing options that all nodes should be expanded.

The system also provided automatic means for managing the hierarchy. When the initial display is computed, all nodes are assumed to be compacted. The system then expands nodes one at a time until an appropriate number of objects exist on the display. When a node is selected, which can be done by name or from another tool through an appropriate message, the system insures that all parents of that node are expanded so the node can be displayed.

The second technique we use, allowing the user to select which nodes should be displayed and which should be ignored, is also quite effective, especially when combined with the use of hierarchy. The system provides three mechanisms whereby the user can eliminate or include nodes. First, nodes can be eliminated from the display using a shift-left button click on the node. If the node represents a hierarchy, the whole hierarchy is removed. Secondly, the user can pick and choose which nodes should be displayed or ignored using the same sequence of dialog boxes used for indicating which should be expanded or compacted. Finally, the user can provide regular expression patterns of names to include and exclude. These patterns can be applied additively (i.e. a name is included if it matches the pattern, but not excluded if it doesn’t match) or completely.

The tools provide several convenience functions for managing whether nodes are displayed or not. It will insure that any current selections and their parents are not ignored so that the selections will be displayed. It also provides menu buttons to include all nodes and to exclude or include system nodes (i.e. those that come from system libraries and not the user’s application).

The final browsing option allows the user to focus on a subgraph of the overall display that is induced by the current set of selections. Here the user can specify the number of levels to consider and whether paths leading into the selected nodes, out of the selected nodes, or both should be used. If the user asks for the local graph, the tools will start with the selected nodes, identify all nodes that are reachable using only the given number of links, and display the resultant subgraph. This allows the user use the overall browsing techniques to find the nodes of interest and then to see subgraph induced by those nodes. A special case of this technique is used in the call graph display. Here, an additional option of only displaying nodes that are reachable from the main program is provided. This is useful when there are significant numbers of nodes that are defined but not used that would otherwise clutter up the display.

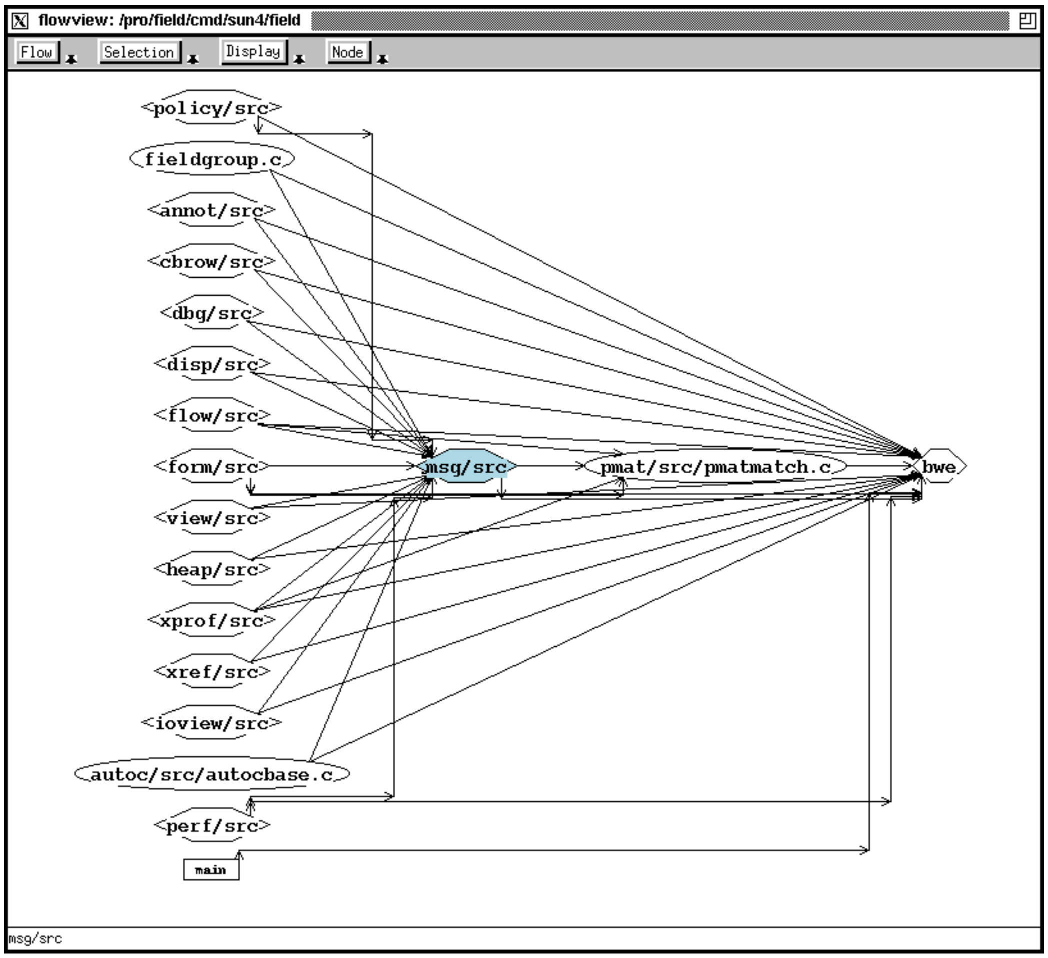


FIGURE 7. Call graph display of FIELD

The overall effectiveness of these browsing techniques can be seen in Figure 7. This is a call graph display of the main FIELD binary and represents about 200,000 lines of C code. The initial display consisted of two nodes, one representing the FIELD directory hierarchy and the other representing the X11 toolkit used by FIELD, BWE. The *bwe* node in the display was left untouched (it represents about five-eighths of the code). The *field* node was expanded to get the various directories shown in the display. Note that where the directory only contained one file (as in pmatmatch.c), the system automatically expanded the directory into the file. Next we selected the main program to cause the corresponding directory to be expanded down to the file level and the main file down to the function level. We then manually removed the other files and functions from this directory. Finally, we selected the node representing the FIELD message server, *msg/src*, and restricted the display to the induced subgraph. The resultant display provides a good overview of the structure of FIELD relevant to the message server. On the left are the various FIELD tools, each of with uses both the message server for communications and the BWE toolkit for its display. The pattern matcher, central to the use of selective broadcasting in the message server, is used by the message server as well as some of the tools.

4.0 Information Encodings

In addition to providing the user with the tools needed for identifying the information of current interest, the FIELD visualization tools attempt to provide as much information as possible through information encodings and associated windows.

Textual information is provided both in the visualization display and in a separate information window on the class and call graph browsers. The status line at the bottom of the diagrammatic visualizers contains two names: the one on the left represents the currently selected item while the one on the right designates what the cursor is currently pointing to. This latter is updated continually as the user moves around the display. The information window is updated each time the user makes a new selection to contain a description of the current selection. This window is active in that the user can click on relevant parts of the display to get additional information or select objects.

Other information is encoded in the shapes of nodes, the line styles and thickness of arcs, using colors, and with text as part of the display. The class browser shown in Figure 3 shows several of these encodings. Color is used both to show the selected classes (light blue) and the selected member (systemCost in EMPLOYEE, green). Color is also used to show the relationship of this member to the corresponding member in other classes. Here pink indicates a redefinition, yellow a inherited instance, thistle (light purple), an instance between this member and its definition in a superclass, orange the defining instance of the member if it is inherited from a superclass, and cyan represents a redefinition. Where a class has been collapsed to only show the name and not the members (as is done here at the user's option for all non-selected classes), the member encoding is used to color the class. The fill style for the class or the class name if the class is expanded, provides additional information. If the class is abstract then it has a halftone fill. If the class represents a collapsed hierarchy, a solid fill is used. The triangles on the left of the member names indicate either method (pointing right) or data (pointing left) and the protection, with solid indicating private, shaded protected, and hollow public. Friend and static members are indicated by a box with an X rather than a triangle. Arc styles indicate either public or private inheritance via arc thickness and friend relationships via dashed lines. Virtual inheritance would be indicated by an arrowhead with a bar. Member names that begin with a double colon indicate inherited members.

Additional information would be encoded and displayed if desired. Different arc styles are used to express the client-supplier relationship, the type relationship, the calls relationship. Members can be displayed with an additional textual field that contains one or more of the key letters I for inline, V for virtual, v for implicitly virtual, P for pure, C for const, S for static, and F for friend. If the user wants members display for all classes, then the selected classes are indicated both by color and by increasing their size.

The other diagrammatic editors used similar techniques to encode the information that is particular to their display. Similarly, the heap and input/output visualizer displays use color and fill patterns. In the heap display, color can encode, at the user’s option, the size of the block, the time the block was allocated, the type of the block (if the information is available), or the source of the allocation. In the input/output visualizer, color can denote either block size or the time the input/output operation was done. Here fill patterns distinguish reads and writes in such a way that overlapping I/O operations can be easily detected.

5.0 Visualization Effectiveness

The visualizations provided by FIELD were designed to accomplish several goals. The simplest was to provide a reasonable, visual interface to the underlying UNIX tools. This is best seen in the *formview* browser which serves as a front end to both *make* and *rcs* (i.e. commands to both of these tools can be issued through the graphical interface). Similarly, the call graph and class hierarchy browsers provided a front end to the cross-reference database, while the data structure display tool provided a partial debugger interface. This goal, to the extent it was attempted, was quite successful.

A second goal was to provide program understanding. This can be divided into two parts -- understanding the dynamic behavior of a system and understanding the static structure. The views that offered insights into the dynamic behavior, have been successfully used and valued for a variety of systems, both large and small. *Heapview* has been used to find memory leaks, allocation anomalies, and related memory problems in large (200K line) systems as well as in understanding the memory behavior of system libraries (Sun’s XGL). The data structure display tool has been widely used in introductory programming classes both to provide an understanding of the student’s data structures and to facilitate object-oriented debugging.

The tools have not been as successful at providing insight into the static structure of a system. While the class browser and call graph displays have been used, they have not been widely used for program understanding. An experiment we ran to evaluate their effectiveness for program understanding showed only a marginal improvement from using the tools. This unintuitive result can be explained several ways. A large part of the problem was speed -- the time it took to generate the data for the visualizations (especially for C++ programs) was such that they could not be used for one-shot questions. Moreover, the time needed to become comfortable with all the features of the tools was a significant barrier. A second problem was the lack of resolution on a 2D display of a large system and the difficulty in achieving the “proper” visualization for a given question. Another problem is that the information conveyed by the diagrammatic browsers was known to the programmer a priori (since it was typically used on their own program), and did not offer much in the way of insights.

The diagrammatic tools were not unsuccessful, however. They have been widely used and relied on as a browsing mechanism within the environment. The tools provide a convenient way of locating code for a particular routine or method — in FIELD, whenever the user clicked on a node in the call graph or the class browser or a file in the dependency display, the corresponding code would be brought up in an editor. This became the primary means for the student programmers for navigating around their programs.

6.0 Conclusions

Visualizations for programming environments are many and varied. FIELD demonstrated a wide variety of visualizations for different applications: as front ends to existing UNIX tools, as visualizations to show the static structure of a system, and as visualizations to show a system in action. FIELD demonstrated the effectiveness of such visualizations and pointed out their weaknesses. It illustrated a variety of techniques for using the limited screen space to display the large quantities of information inherent to a programming environment, both in terms of browsing and in terms of information encoding.

Work on visualizations for programming environments is continuing. Current work by this researcher and others includes efforts aimed at providing a high quality textual and graphical hyper-linked interface to all software artifacts, from design documents, to code, to user interface diagrams, to static and dynamic visualizations. Other work involves extending the 2D visualization framework offered by FIELD to three dimensions, both to make more effective use of the display and to convey additional information through depth and the additional spacial relationships that are available in 3-space. Another effort is aimed at addressing one of the weakness of the diagrammatic visualization tools of FIELD by providing the programmer with a high-level, interactive visual query interface for quickly defining both what information should be visualized and how it should be displayed.