Interacting with the FIELD Environment

Steven P. Reiss

Brown University
Dept. of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-89-51
May, 1989

Interacting with the FIELD environment

Steven P. Reiss

Department of Computer Science
Brown University
Providence, RI 02912

May, 1989

Abstract

This paper describes the ideas behind and the use of the FIELD programming environment. FIELD is an integrated environment that is based on UNIX programming tools and that runs on top of the X11 windowing system. It provides a consistent and complete interface to its tools. In addition, FIELD provides a variety of tools for program and data visualization. The key concepts behind FIELD are an integration mechanism based on a simple, central message server, an annotation editor for integrating access to the source file with the other tools, and the use of a high-level user interface toolset.

Keywords: Programming environments, user interfaces, program visualization.

---

† This research was supported in part by grants from Defense Advanced Research Projects Agency, the National Science Foundation, and the Digital Equipment Corporation. Equipment support was provided by the National Science Foundation.

Interacting with the FIELD environment

Steven P. Reiss

1. Introduction

This paper describes the FIELD programming environment from a user's perspective. FIELD is a programming environment developed at Brown University that attempts to integrate a rich and powerful set of tools to make full use of the capabilities of today's workstations. FIELD uses simple integration mechanisms to combine the standard UNIX programming tools: editors, compilers, debuggers, profilers, and the make facility, with our own tools for program and data visualization. This paper shows how FIELD provides a complete programming environment to the user. It describes the design strategies and principles involved in building the environment. Moreover, it proves that the simple mechanisms used by FIELD can provide a highly integrated environment to the user.

Several design goals influenced the development of the FIELD environment. The environment was designed for both teaching and research. It had to provide a programming environment that was user-friendly, simple, and obvious enough to be used by students in an introductory programming course. At the same time, it had to be powerful and efficient enough to be used by advanced researchers writing moderately large (100,000 line) programs. Along with these direct goals, we wanted FIELD to provide a showcase for some of the research done at Brown on program visualization and to serve as a research vehicle for studying new tools and new approaches to programming and visualization. In addition, we wanted to develop the environment quickly and with a minimum amount of effort.

The key concept used in the FIELD environment is the integration mechanism we call *selective broadcasting*.[14] FIELD starts with a set of independent tools. These tools communicate through a central message server using discrete messages represented as strings. Each tool, at startup, registers patterns with the message server describing messages it it interested in. Then, as the environment is used, tools send messages to the server. The server matches these incoming messages with the patterns registered by

the various tool clients, and selectively rebroadcasts the messages to those clients that have matching interests.

The message facility solves many of the fundamental problems involved in putting a programming environment together quickly and in integrating a diverse set of tools into an open framework. Other problems arise in building the user interface. The primary objective of this user interface is to make the environment look and feel like a single, integrated system. This is done using a single user interface toolkit consistently.

A second problem related to the user interface is providing consistent access to the source code. Most of the tools in a programming environment reference the source code of the system. In a standard UNIX environment, the user views the source differently in the debugger, the editors, the cross-reference facilities, and so on. In FIELD we wanted to provide consistent source access for all tools using a single view. We do this using an annotation editor, a standard editor where annotations can be attached to each line. The annotations are used with the message passing mechanism to allow the user to interact with other tools using the source and to let the tools interact with the source. Messages sent by other tools can add or remove annotations; a user request to add or remove an annotation is translated into messages that serve as commands for other tools.

The FIELD user interface was built using the workstation tools we have been developing over the past seven years in the Brown Workstation Environment.[8, 12] These tools offer considerable functionality while greatly simplifying the job of the programmer who has to use them in an interface. The particular user interface tools FIELD uses extensively are:

- **EDT** -- a powerful base editor that offers over a hundred different editing commands with user-configurable bindings, and that can be run in any window in a variety of different configurations;
- **GELO** -- a combination of packages that provides user-definable graphical displays of structured data;[10, 13]
- **HELP** -- an interactive, hierarchical help package;
- **LEAF** -- a geometry package that allows windows to be quickly laid out;
- **STEM** -- a set of menu packages that offers pull-down menus, static menus, dialog boxes, and scroll bars;

- **WIND** – an in-process window manager that provides a control panel of available windows and that provides windows adorned with buttons for window manipulation.

FIELD is careful to use these tools consistently to provide an appropriate user interface for an integrated programming environment.

The FIELD environment also emphasizes program and data visualization. The environment supports the automatic visualization of user data structures, including dynamic updating of these structures while the program executes. Program execution can be viewed through the source or through visualizations of the source such as a call graph. In addition, the system provides hooks for algorithm animations that can be designed as part of a course lecture or that could be used by students or researchers for understanding and animating their own programs.[17]

The next section provides a brief review of current efforts in providing program development environments and shows how the FIELD environment fits in. The next section attempts to provide a sense of the FIELD environment by demonstrating its use on a simple program. This is followed by a detailed discussion of the messages used in the environment and the annotations that are provided. Finally, we conclude by discussing the strengths and weaknesses we have found in the current framework and possible directions for future research with the FIELD environment.

2. Program Development Environments

Integrated programming environments have been widely touted as a means for increasing programmer productivity. Moreover, the controlled and understandable environment they offer the programmer is ideal for instruction. A wide range of integrated environments have been developed over the past twenty years.

Most of today's environments offer independent tools that are integrated through the file system. UNIX is an example of such an environment. In UNIX, programmers edit their source files. They use SCCS or RCS to check files in and out and to manage versions of files. They run the UNIX *make* program that takes a file describing how the system is to be built and runs the appropriate compilers and

loaders on the source files to produce object files and the resultant systems. The debugger runs on the binary file and accesses the source files as appropriate. These tools are integrated only in the sense of operating on a common set of files. They each have different interfaces and obey different conventions. Programmers are free to choose the set of tools they want to use. None of the tools takes advantage of the others or interacts except through their eventual outputs in the file system.

There have been some attempts to extend UNIX's limited degree of integration. Sun's *dbxtool* provides a make command to run the UNIX *make* program and an edit command to run an editor on the appropriate source file.[6] The GNU Emacs editor can parse the output from the compiler and go to lines containing errors.[16] The default make rules have been extended to do many SCCS functions automatically. All these extensions, however, are ad hoc and involve simple extensions to one particular tool. They do not provide a common interface nor do they offer the user the feel of an integrated environment.

More integration can be found in single-language environments where all the tools are combined in a single system. The tools generally share a common interface and operate on a single program representation. Such environments exist both with languages such as Lisp and Smalltalk, and with small-scale systems on personal computers.

These systems can achieve a high degree of integration. The PECAN system,[9] for example, allowed multiple editors on the source and updated them all as the source changed. It used the source views to animate program execution, and provided an incremental compiler that ran as the editor detected a source change. The compiler in turn updated a set of semantic views such as the symbol table. It also detected errors and highlighted them in the source views. Breakpoints could be set in any of the source views. More recent systems include examples of highly integrated, single-system environments that are aimed at students such as MacPascal or Lightspeed Pascal,[19] as well as research environments such as CENTAUR.[1]

Such systems, however, are typically complex closed environments that fail to take advantage of existing tools. Languages such as Lisp and Smalltalk allow an extensible environment and direct access

to the execution framework. This permits new tools to be built incrementally on top of the system, so that with time, the system gains functionality and complexity. Traditional languages such as C or Pascal are not amenable for use in such an extensible environment. Most of the environments developed for these languages use an interpreter to achieve a high degree of integration between an executing program and the rest of the system, a strategy that generally limits these systems to handling small programs. An exception here is the SABER C system that uses a C interpreter that allows mixing interpreted and compiled code.[15]

Much of today's programming environment research focuses on environments that are based on a central database system. Here all the tools are designed with the database in mind. All the information about the program is stored in the database. The tools access the database to get the information they need about the program or from other tools. This approach offers many potential advantages. The tools can be highly integrated as in a single system since they share data structures through the database. The tools can also share processing. For example, a parser could store a syntax tree in the database that would then be used by the compiler, the cross-referencer and a syntax directed editor. However, the use of a program database in general has disadvantages. The database system itself adds substantial complexity to the programming environment. Database systems are generally large, complex programs and a program database that deals with multiple clients and maintains consistent information about a program is no exception. It is difficult to fully integrate existing tools or tools that were not designed with the database in mind into such an environment. Moreover, because the representation of the program must be understood before most of the tools are written, adding new tools that do not fit well with the original *a priori* definition can be difficult. This is the approach taken with the proposed Ada environments,[7] in the PCTE efforts,[2] and in the Arcadia project.[18] Other efforts attempt to combine a database-oriented environment with existing systems. PROCASE's SMARTsystem, for example, uses a database program representation for editing, building and debugging, but dynamically generates source files for the system compiler.[3]

FIELD provides a third alternative, using its loosely-coupled message facility to provide tool integration. This alternative offers several advantages, primarily simplicity, ease of reuse of existing

tools, and ease of extensibility. This, combined with the annotation editor to provide consistent access to the source throughout the environment, leads to a powerful programming environment that can effectively use existing and future tools. This approach simulates a tightly coupled environment through the communications mechanism, and uses the file system where tight coupling is not required. Our experience, as we show in the next section of this paper, is that this loosely coupled approach can offer the user an environment that seems tightly integrated. This experience has also been verified commercially where an early version of FIELD served as the conceptual basis for Hewlitt-Packard's Softbench system, another integrated collection of tools based on a message server.

3. A Tour Through FIELD

The FIELD environment consists of independent tools that can be invoked in a variety of ways. The tools can be configured as independent UNIX processes, with all tools in a single UNIX process, or, as any combination in between. FIELD takes advantage of this by providing a variety of different invocation options. First, it is possible to run each tool independently, starting them each at the shell level. Secondly, there are several default configurations of FIELD that offer logical combinations of two FIELD tools such as the debugger and an accompanying editor. Finally, the whole environment can be run with a control panel to allow the user to dynamically invoke any of the tools. Other configurations are possible, being defined by a resource file at system startup. For example, this is used in a simplified version of the system for novice programmers.

For our tour, we want to illustrate the whole environment and thus we start by running the full system. This causes a window containing the control panel, shown in figure 1, to be created on the user's workstation. At the top of this window is a pull-down menu bar. Most FIELD windows contain such a menu bar. These menus are used by clicking on them with the mouse and operate substantially like Macintosh menus. The central portion of the control panel contains icons representing different commands or tools that are available within the environment. Both the pull-down menus and the tool icons are configurable. Each FIELD tool can be tied to one or more menu buttons or icons for its initial invoca-

tion. If the tool is later iconified, then a corresponding new button will appear in the menu or panel region from which the tool was initially created. These commands can also include strings to be executed by a UNIX shell or calls to dynamically loaded user functions. The standard layout of the FIELD control panel in figure 1 has commands such as access to the help facilities, the ability to set the working directory, and quit, located in the *Command* menu, and non-programming UNIX tools including a shell window and an editor window, in the *UNIX* menu. Access to all the FIELD tools is through the buttons on the control panel.

To use FIELD, we first request an annotation editor to display the source file and a debugger window for the executable by clicking on the corresponding icons on the control panel. Annotations are the primary mechanism for interacting with the source within the FIELD environment. Annotations for error and warning messages, the current focus of the debugger, the current line of execution, and the cross-

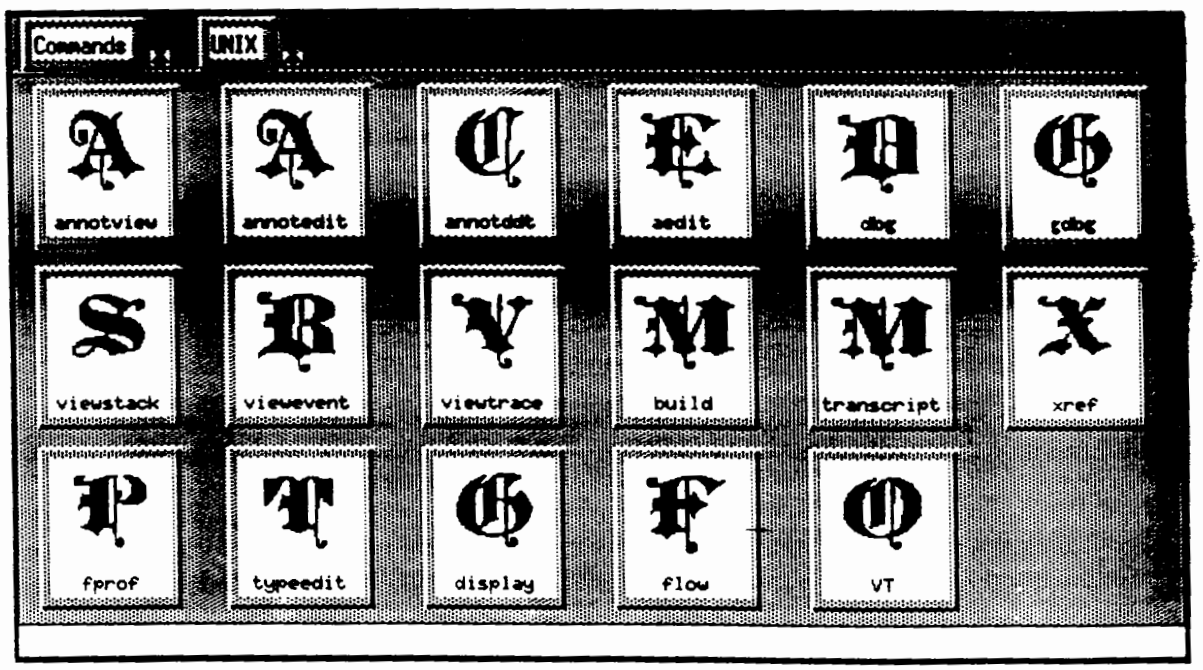

Figure 1: The FIELD control panel

reference focus are all information annotations that are created when other tools in the environment broadcast information relevant to the source file currently displayed. In general, each particular type of annotation can be defined or deleted by a particular message coming from the central message server. These annotations can also be monitored. A message requesting the addition of a monitored annotation will cause the editor to change the file or line being displayed so that the annotation will be visible to the user. In addition, annotations such as BREAK representing a breakpoint allow the user to interact with other tools of the environment using the annotation editor. When the user requests the addition of a breakpoint, a message associated with the BREAK annotation is sent to the message server. Here the message is a command to the debugger to set a breakpoint in the given line of the given file. The debugger receives this message, sets the breakpoint, and then broadcasts another message informing all tools that a breakpoint was set. It is this message that causes the BREAK annotation to appear.

Several different annotation editors are available. *Annotview* is used with the compiler and the cross-referencer to view errors or references in source files. *Annotddt* is used with the debugger. *Annotedit* is the general purpose file editor with annotations and *aedit* is a file editor without annotations. These are instances of the same tool, but they monitor different sets of annotations. Thus, *Annotview* will change the file and line being displayed to show the most recent compiler error message or cross-reference request, while *Annotddt* will change the file and line to show the current debugger focus or line of execution. The user can dynamically change what annotations are available within a particular annotation editor and what annotations that editor is monitoring.

The *Annotddt* annotation editor on our source file, tree.c, is shown in figure 2. This window is created and maintained by the WIND package of the underlying environment. It illustrates several features of the user interface. First, it shares a common set of buttons with all the other FIELD windows. These can be seen in the top and bottom panels of the given window. The iconic button in the upper left corner is for moving the window on the screen. The iconic buttons in the other three corners are for resizing the window. The words *Remove* and *Push* in the bottom panel are buttons to remove and change the stacking order of the window respectively. In addition, the user can click in either the title bar or the

blank space in the bottom pane to pop the window to the front of the display.

Beneath the title region is the menu bar for the annotation editor. This contains buttons specific to the annotation editor on the *Annotate* menu, and commands such as *Compile* and *Make* on the *Commands* menu. The remaining three menus, *File*, *Edit* and *Move*, are generic menus provided by the EDT editor. The body of the annotation window is divided into four panels. The panel on the left displays the annotations. One annotation is currently displayed: an pair of eyeglasses showing that the debugger is currently focused on the first line of the main program. Multiple annotations can be associated with a given line. If more annotations exist than fit in the annotation panel, then the user can use the *Rotate* button on the

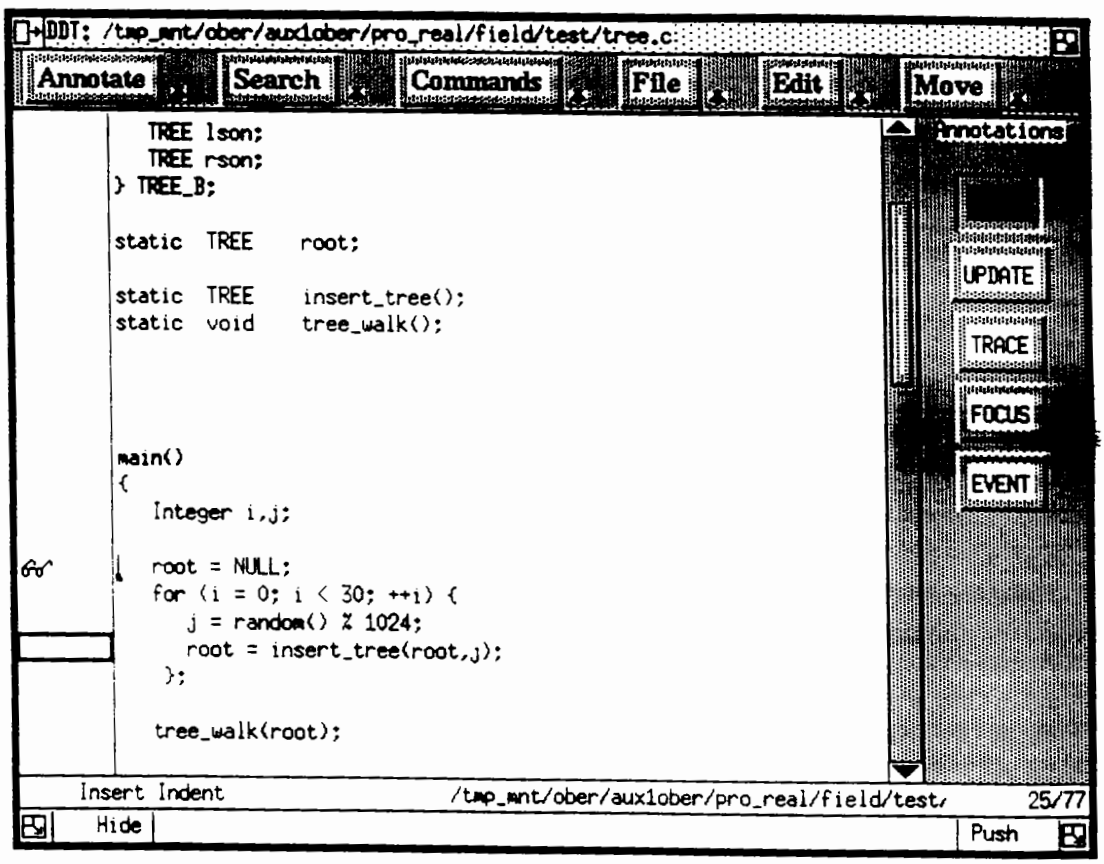

Figure 2: The annotation editor

Annotate menu to show the remaining ones. The rectangle in this region represents the highlighting that shows the annotation area under the mouse. The portion of the main window on the right contains a static menu listing the set of annotations the user can explicitly request. It shows that an annotation request (the user clicking with the mouse in the annotation area) will result in the addition of a breakpoint annotation. The other two panes of this window, the central part containing the program text along with the scroll bar and the status display underneath it, are provided by the EDT base editor.

In addition to putting up a window on the source file, we opened a debugger window on the binary image. This window is shown in figure 3. It is adorned in the standard way and contains a menu bar. The *Buttons* menu allows the user to add, remove, change, load and store the mouse-clickable buttons in the lower part of the window. The remaining menus again are provided by the EDT editor. Here EDT provides a complete interactive transcript of the debugging session. The user can browse through the previous output and can use editor commands to build the next line of input. The two boxes at the bottom of the window contain the current debugger focus and current line of execution. The region above these contains user-definable buttons for the most frequent debugger commands. These can be set initially by the user and can be changed dynamically. Moreover, the text they echo to the debugger can contain references to the current editor selection, the current line of execution, the current function, or to user-settable parameters that will be requested using a dialog box.

Figure 4 shows the full screen of our interaction with FIELD a little further on. Here we have added a breakpoint by clicking on the appropriate line in the annotation editor, and have started the program executing by clicking on the *Run* button in the debugger view. The annotation editor displays three annotations: eyeglasses denoting the current debugger focus, an arrow highlighting the current line of execution, and a stop sign for the breakpoint. The window on the lower right is a stack viewer. It catches and formats messages from the debugger to display the current contents of the run time stack. The window above it is an event viewer. It catches and formats messages describing the existence of breakpoints and similar execution-related events. These windows share the same adornments and menu bar of the other FIELD windows.

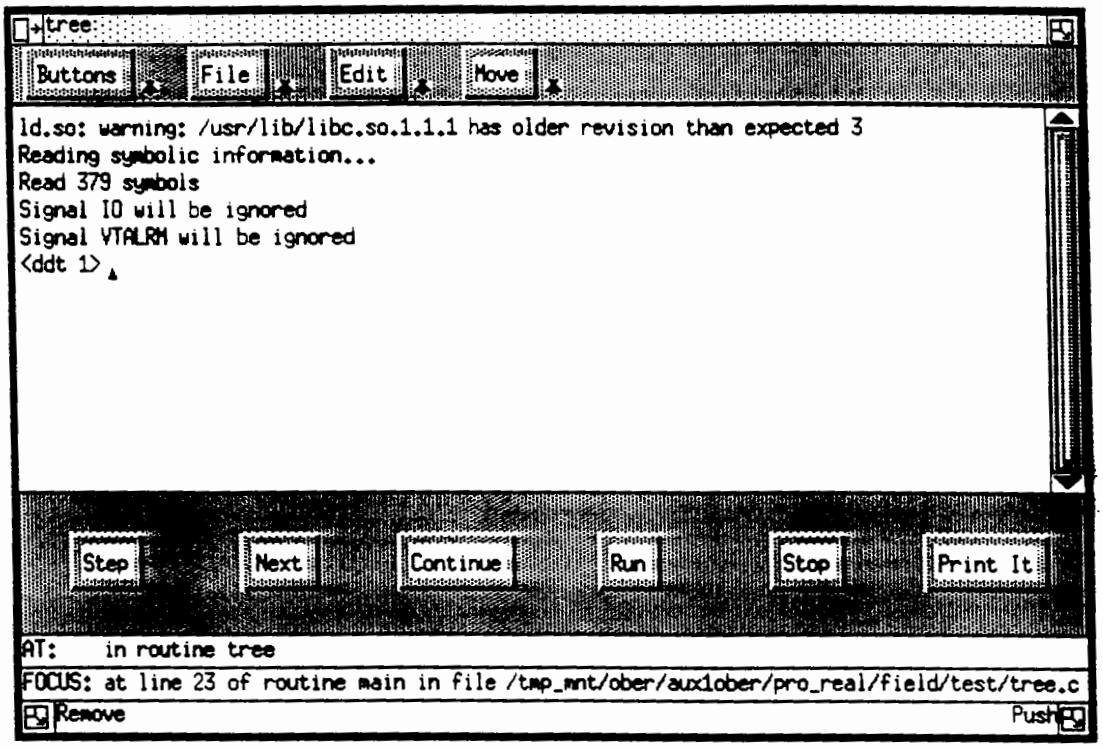

Figure 3: The debugger interface

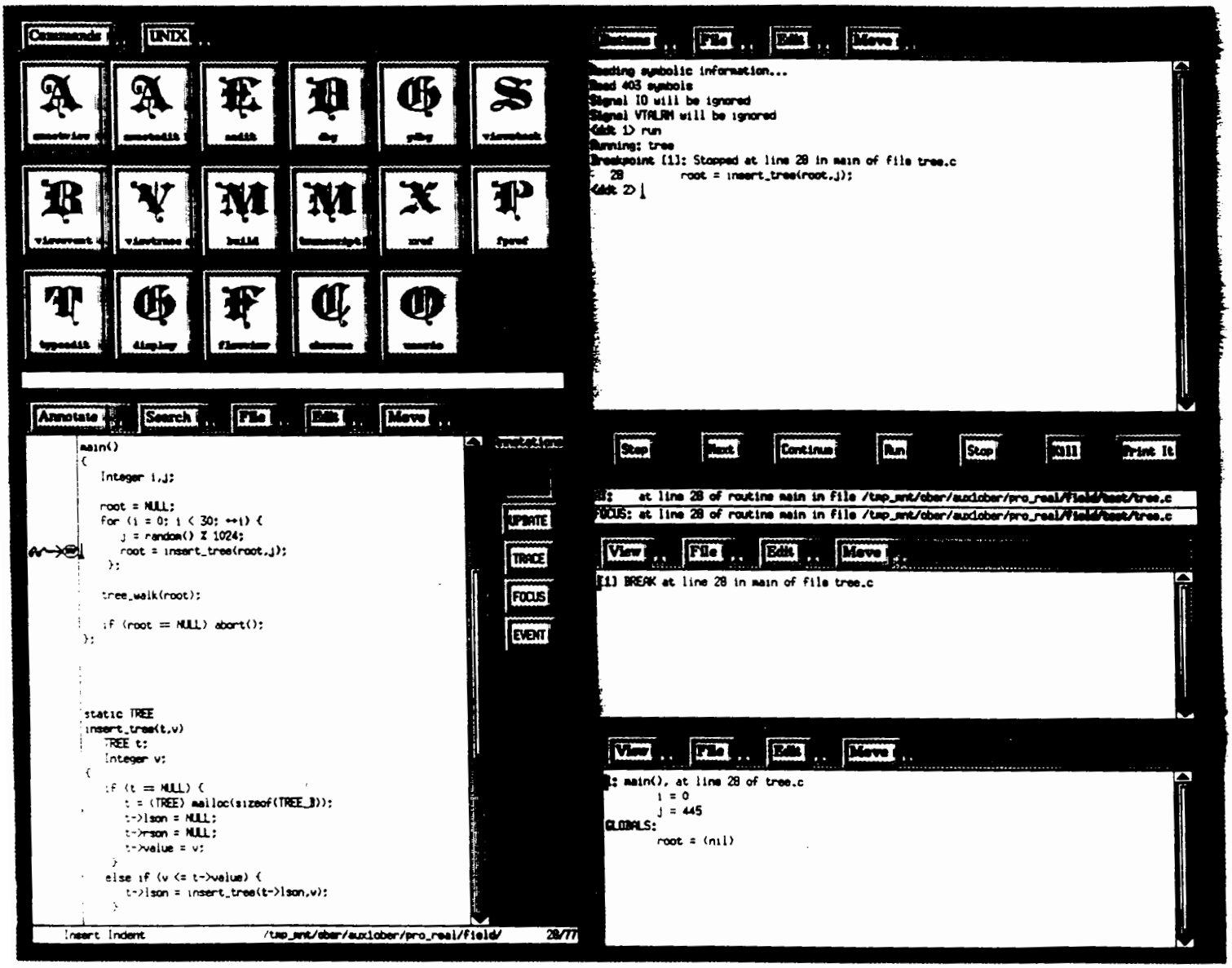

Figure 4: The complete environment

The annotation editor can also request compilation. For example, in figure 5. we have edited the source file slightly and selected the *Compile* command off the *Command* pull-down menu. This sends an appropriate message the the make interface described below. It has the effect of putting up an information box telling the user that the system was compiling followed by a dialog box that contained the compiler output when the compilation was completed. After removing this dialog box, the annotation editor, shown in figure 5, includes new annotations corresponding to the error messages. The user can click on these annotations to view all the error messages at the corresponding line.

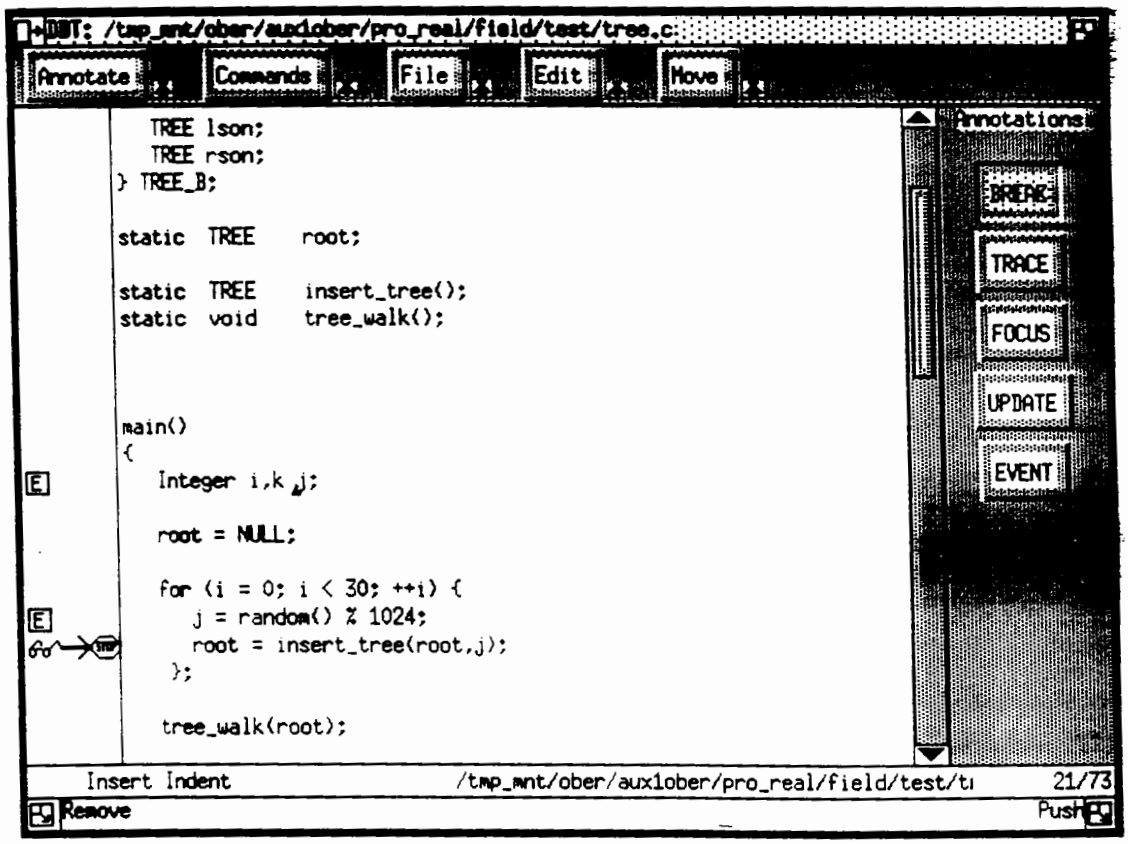

Figure 5: The annotation editor after compilation

Other tools are available within the FIELD framework. One, shown in figure 6, provides a hierarchical call graph display of the program. This tool uses the underlying cross-reference facility of FIELD (by way of the message server) to build a complete static call graph. It organizes this call graph hierarchically, using the inherent function-file-directory hierarchy of UNIX programs, and displays a reasonable portion of the result. The user can then select what is displayed, both by ignoring different items and by varying the hierarchy level of other items. Our sample program is simple, consisting of only three functions, and hence does not make use of many of these capabilities.

The call graph display of figure 6 is generated using the structured display tools of the Brown Workstation Environment. This package allows the program or the user to define how a typed structure should be displayed. Here the call graph is displayed as a layout where nodes can either represent func-

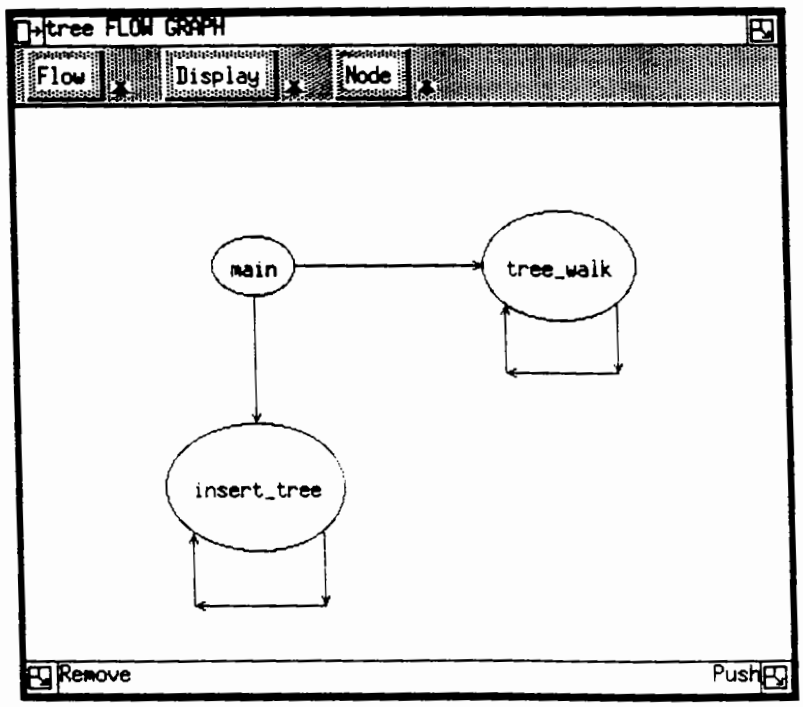

Figure 6: The call graph display

tions (circles), files (boxes), or directories (hexagons), and where arcs represent calls. The layout of the graph is handled automatically by the underlying tools. In addition, the tools provide for user interaction. Thus, the user can select any of the structures being displayed, can request information on the files or arcs, or can request that a FLOW annotation be sent for a particular node or arc so that an annotview editor will display the corresponding code.

The flow graph display obtains the call graph from a cross reference database FIELD maintains for the systems under development. This database can be accessed directly through the cross reference view, XREF, shown in figure 7. The underlying database is a relational database containing relations describing the scopes, references, declarations, calls, functions, classes, members, and files of the system. It supports a relational calculus query language. The user interface provided by XREF is simpler. The *Xref* pull-down menu allows the user to select a relation and then provides a dialog box where whatever information is known about the fields of the relations can be specified. Once the dialog box is accepted, the interface forms a query and displays all tuples from the relation matching the specified information. This interface uses the current editor selection as a default value in the dialog box. For example, figure 7 shows the output of two queries. First we selected the string *insert_tree* in the annotation editor. Then we selected the *Call* relation off the *Xref* menu. The resultant dialog box had the editor selection as the default value for the routine being called and no value for the other fields. We accepted this dialog box, and the resultant output shows all calls to *insert_tree* in the system. Next we selected the string *root* in the annotation editor. We selected the *Reference* relation off the *Xref* menu. This dialog box allows us to further specify what we want references on. In particular, it allows us to restrict the search based on name patterns, on the file or line number of the reference, and on whether the reference is an assignment. Here the name is filled in from the current editor selection and all we did was to accept the dialog box. The output of the query shows all references to the token *root*, with stars denoting assignments. The XREF tool also allows the user to shift-click on the output from any of the queries to display the corresponding code in an appropriate annotation editor.

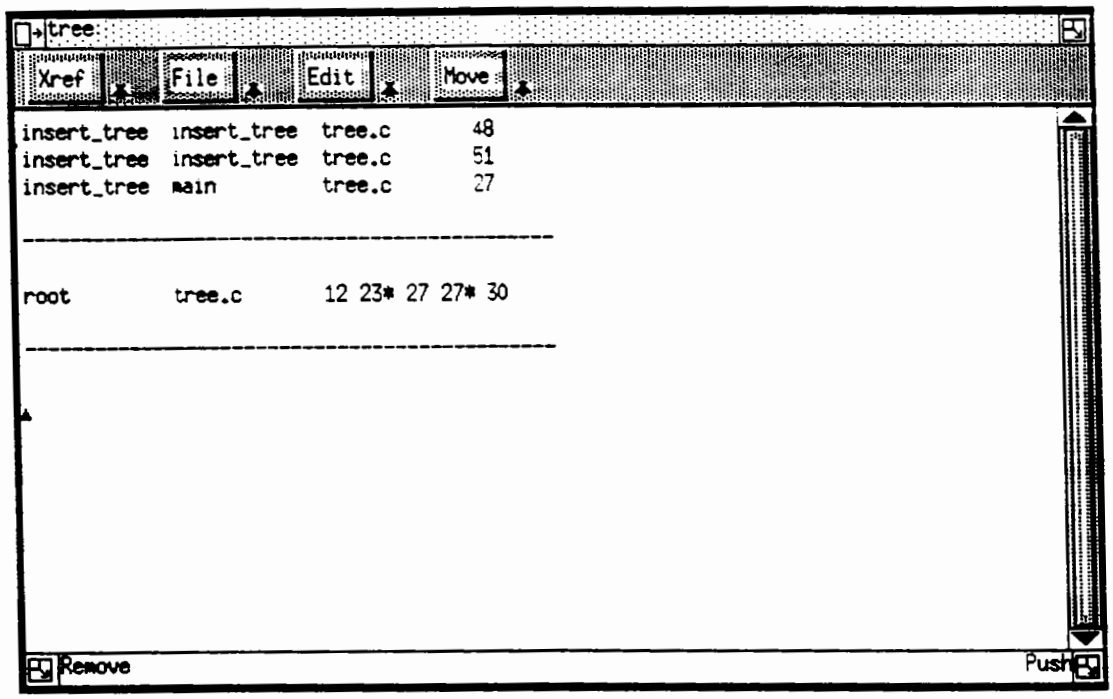

Figure 7: The cross reference interface

Another tool provided by the FIELD environment is an interface to UNIX *make*.[4] Make is available to the other tools of the environment through a message-based interface, used, for example, when requesting a compilation from the annotation editor. In addition, the BUILD tool shown in figure 8 provides a visual interface. This interface reads in standard UNIX makefiles, allows the user to edit the makefiles interactively, and writes out the result as a standard makefile. It uses the user's current makefile as well as a set of default rules to determine all explicit and implicit dependencies and build rules. It allows the user to view and edit this configuration information interactively. For example, figure 8 shows an information display for building the binary file tree from our source file. The user can add explicit dependencies by clicking on the word *Dependencies* and selecting files from a list in a dialog box. Dependencies can be removed by clicking on the file name whose dependency is to be removed. The build rules can be edited in the EDT editor at the bottom of the window. In addition to defining rules and dependencies for

files, BUILD allows the definition and use of macros and the setting of all standard make options interactively. It also can provide a display of the transcript of all builds that the user has done or a display of the dependency graph for a given file.

FIELD also includes an interactive help facility using the corresponding BWE tools. The user can hit the help button on the keyboard at any time. The HELP facility will then automatically provide help information on the window, menu, or button that the mouse currently points to. This information is displayed in a HELP window shown in figure 9. This display contains a STEM menu on top that shows the parent item (*Field*) and children items of the item that help was requested on. The user can select any of these items to navigate through the help hierarchy that has been defined for FIELD. The bottom of the window contains a read-only EDT editor that displays the text of the help information for the current item. A second help facility is shown in figure 10. This is a display of what the three mouse buttons do. It is also a part of the BWE HELP package. Its display changes automatically as the mouse moves

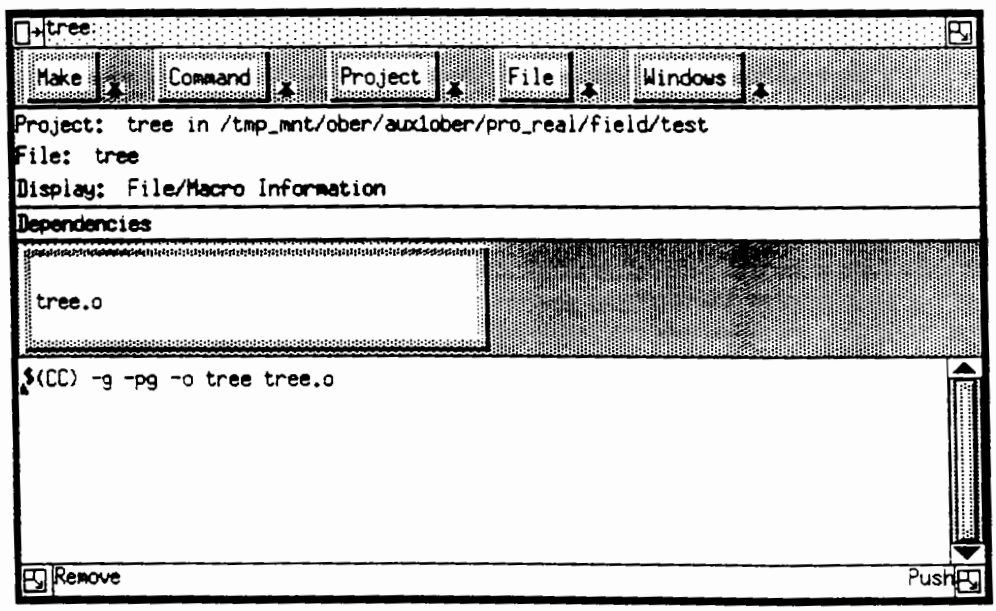

Figure 8: The make interface

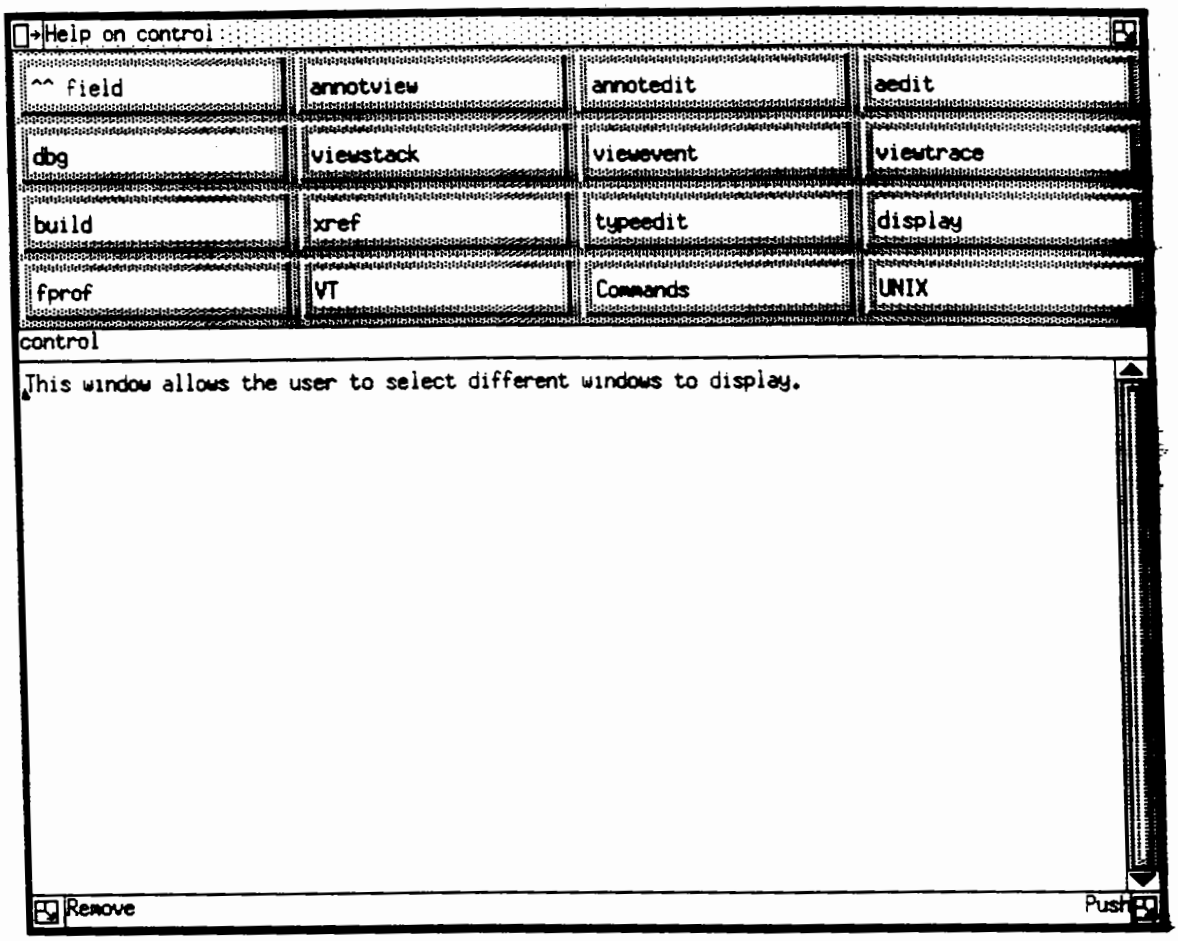

Figure 9: The help facility

around and into different windows.

FIELD provides an interactive interface to the UNIX profiling facility gprof.[5] This tool displays the run time spent in each source file. The user can click on any file displayed, and the tool will display the run time and related information about each function in that file. The user can again click on any function to see the run time broken down on a line-by-line basis. We plan to further develop this tool to offer a graphical view of the profile.

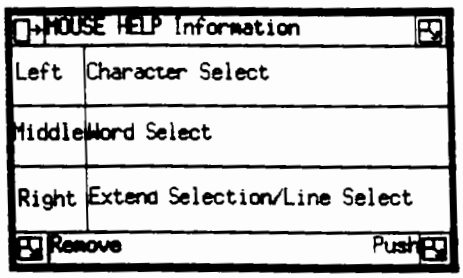

Figure 10: **The mouse help facility**

The last major tool offered by the FIELD environment is a data structure display facility. Our example program does tree insertion. We can put up a display on the tree it creates by asking to display the variable containing the root of the tree, *root*. The tool offers default displays and also allows the user to define the display. After starting the program, and continuing at the breakpoint we previously inserted several times, we have built a partial tree. We then use the display tool to display the current value of the tree. Two different default displays are available. The first, shown in figure 11, uses nested boxes to display the substructures. Each level of the tree is displayed with a top line containing the type name, a side bar on the left to indent the fields, and the fields in turn. This type of representation is suitable for a variety of structures, but is somewhat confusing for the tree we are viewing. The other default display, using boxes and arrows, is shown in figure 12. Here each box is displayed as before, but without the side bar and with fields containing pointers displayed using an arrow to the structure they point to. The whole diagram is then laid out using the automatic layout facilities of the GELO package of the user interface toolkit.

The display tool offers a front end to GELO. It uses STEM to provide the menu bar at the top of figures 11 and 12 and the scroll bars on the r ht and bottom. Under the *Display* menu, the user can zoom, emphasize single items, and change the drawing parameters. The scroll bars allow the user to pan over the zoomed-in view. The *Inset* menu allows the user to take any portion of the display or another

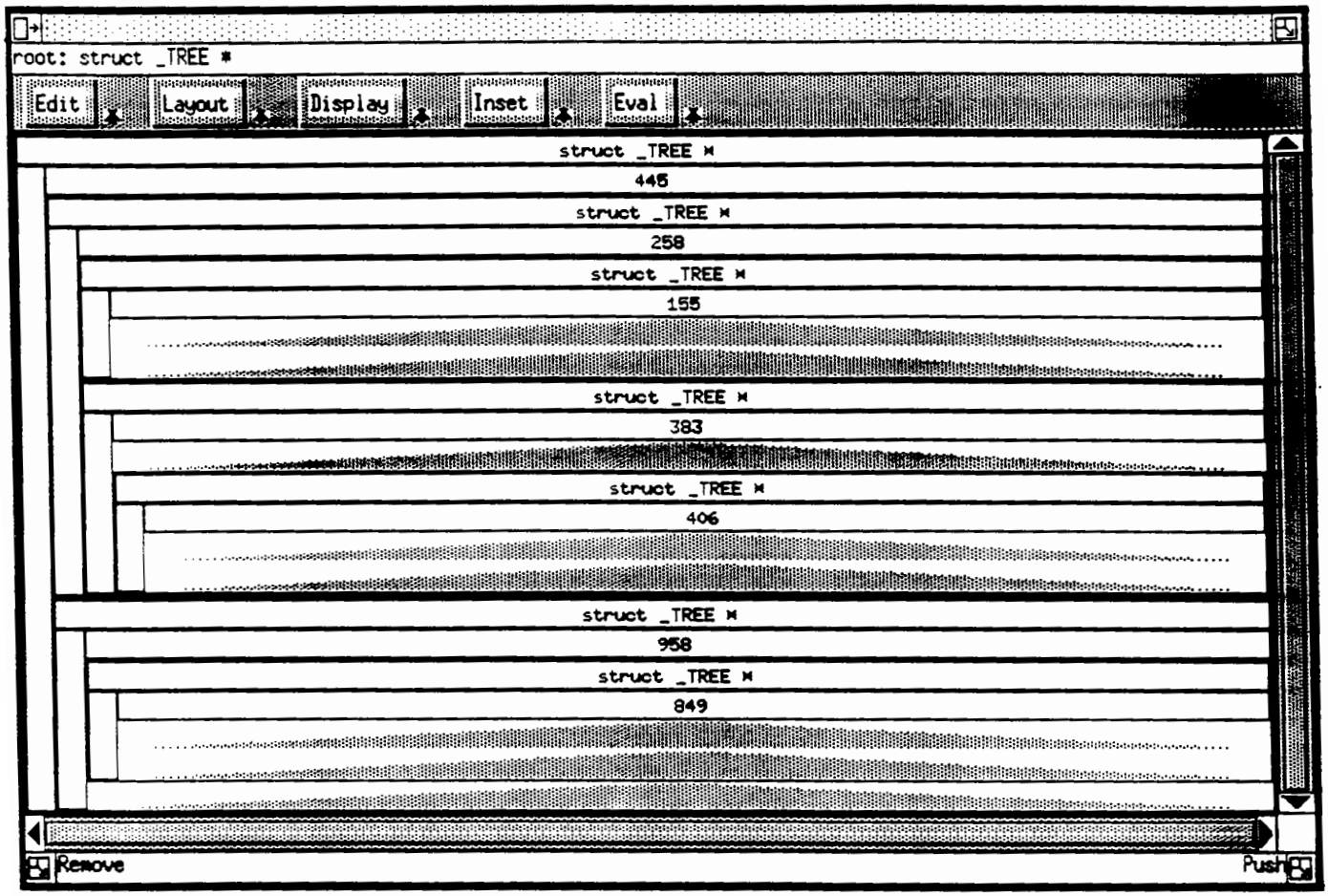

Figure 11: A tree displayed using the default nested method

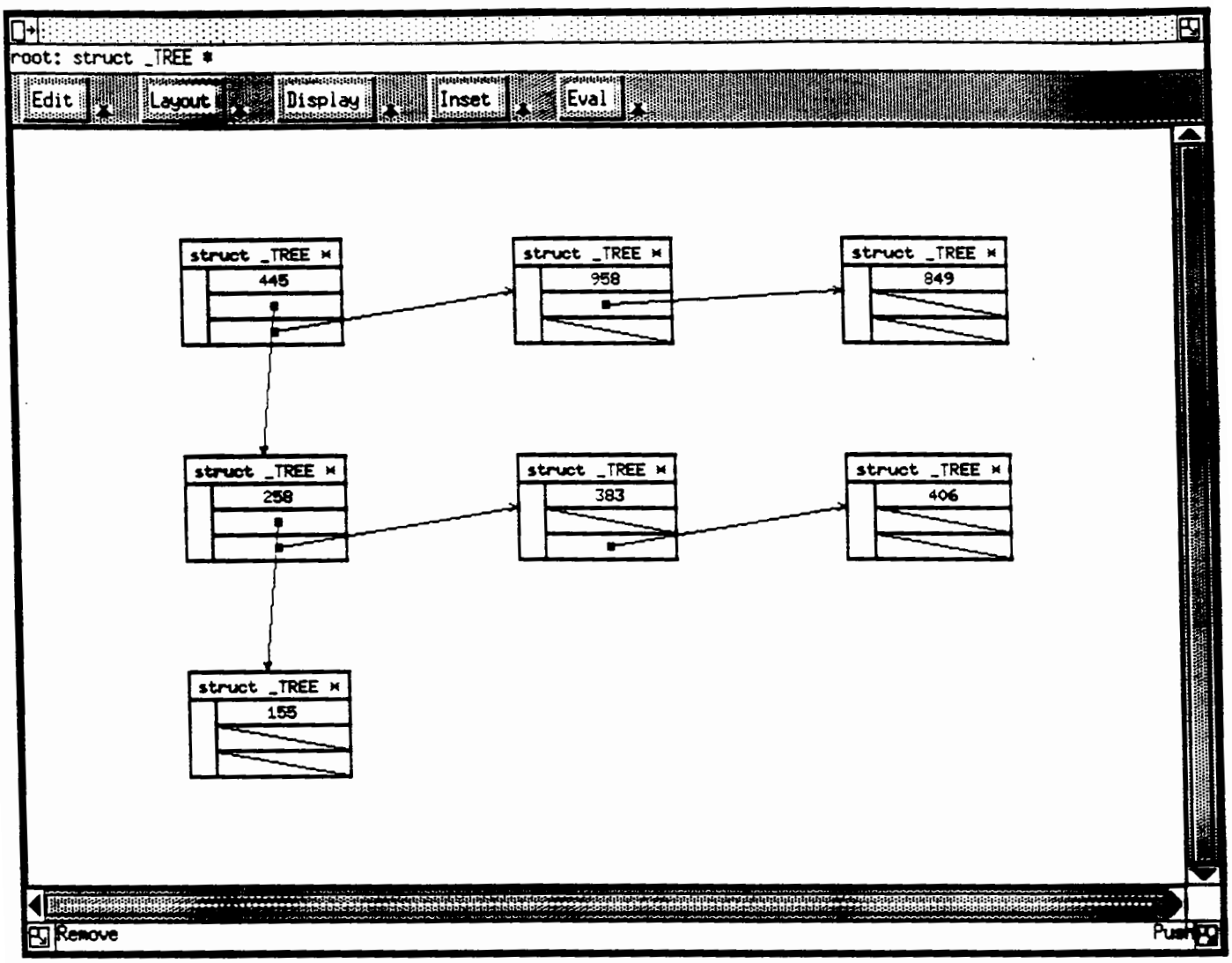

Figure 12: **A tree displayed using the default layout method**

structure and put up an inset window in the lower right showing that item. The *Edit* menu allows the user to edit the displayed data structure graphically, changing values as appropriate. These value changes are mapped back to changes of the underlying data structure in the program.

In addition to the default data structure displays, FIELD allows programmers to define and save their own methods for displaying particular structures. For example, the user can have the trees of this

sample program displayed like trees by creating the layout definition in figure 13. Here the display is partitioned into a rectangular layout with a box for the value of the node on top, and the two subtrees displayed recursively using similar layouts underneath. Not shown directly in this figure are user constraints to set the value to a standard size and arcs to be drawn from the value to both the lson and rson boxes. Another diagram was drawn to tell the system to display an empty tree as a fixed size triangle. The result of these efforts is the tree diagram shown in figure 14.

The data structure display facility is tied into the rest of FIELD. It normally relies on manual updating by the user. That is, the system will leave the current picture of the data structure intact until the user

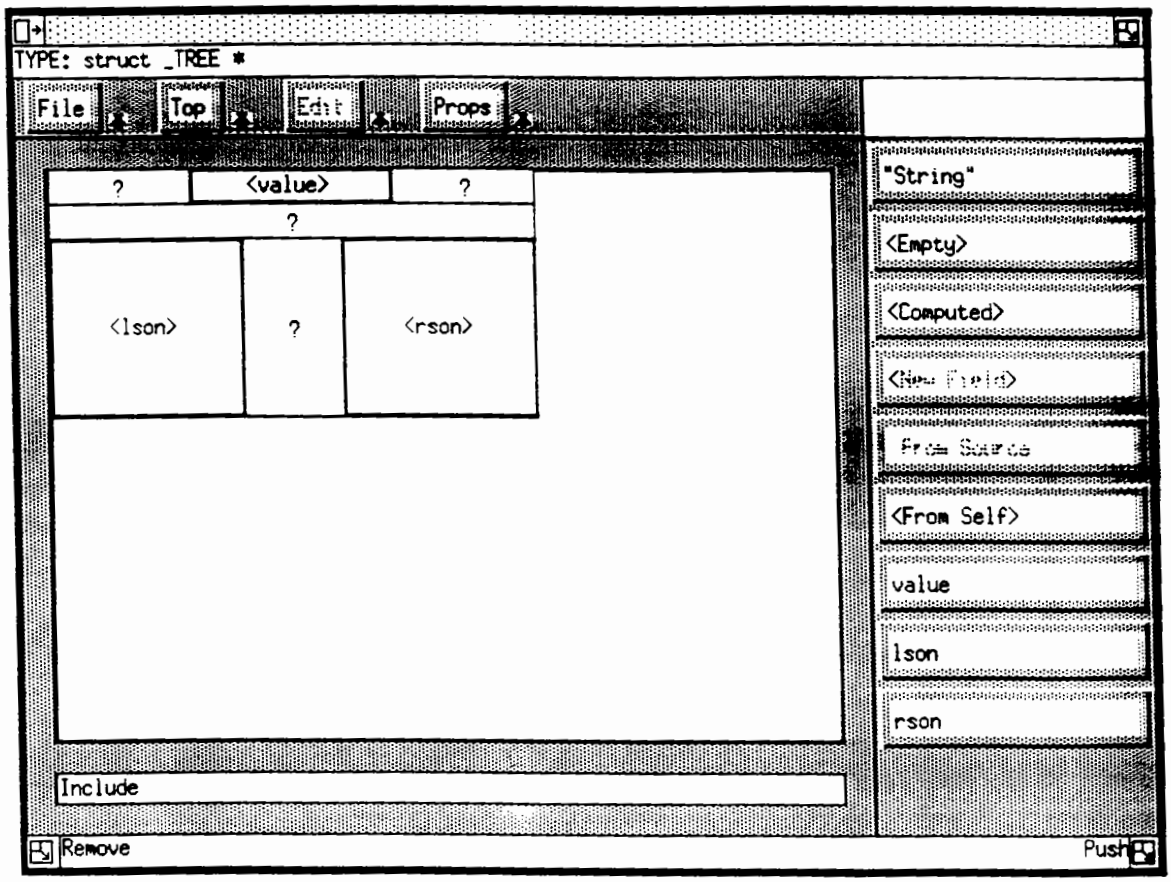

Figure 13: The type display editor

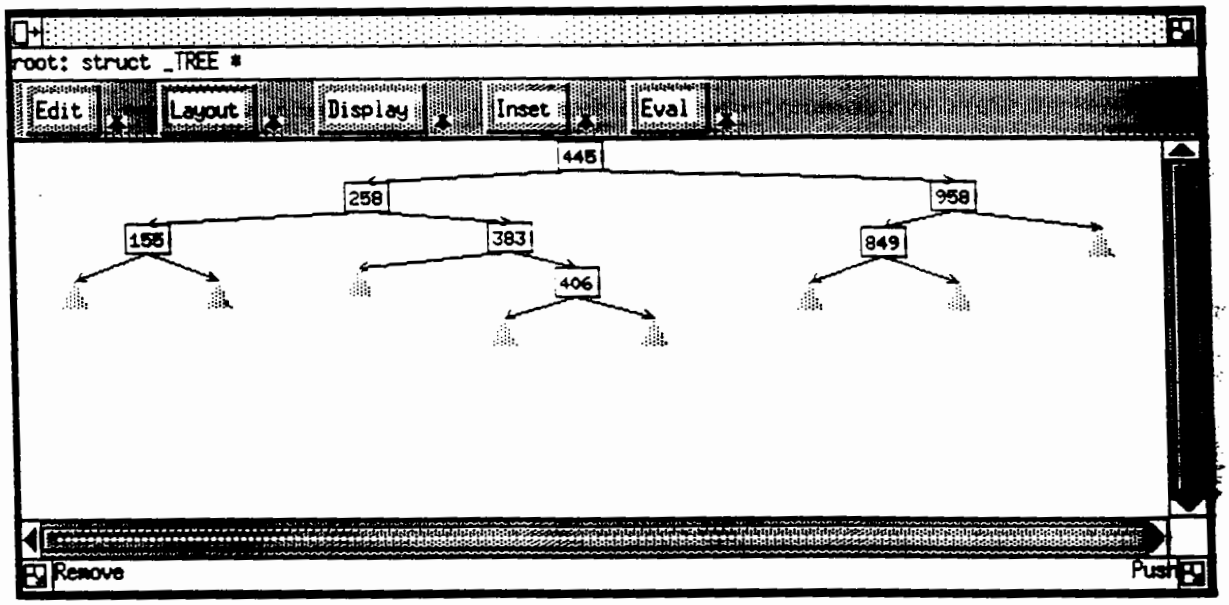

Figure 14: The tree displayed using user-defined methods

specifically requests that it be updated to reflect the current values in the program. As an alternative, the user can request that the system always update the displays when the debugger gets control or that the display be updated whenever control reaches designated points in the program.

4. Messages

The FIELD message facility is based on a central server communicating with multiple clients via Berkeley UNIX sockets. All messages are passed as strings. Each client registers patterns that describe the messages that it is interested in. The server matches incoming messages against the previously registered patterns and selectively rebroadcasts these message to all clients whose patterns match. Messages can either be sent asynchronously without a return value, or synchronously with one. Synchronous messages return control to the sender after all clients that have matching patterns have responded and return the first non-null string value returned by a client. A more detailed description of the message server and its associated utilities can be found elsewhere.[14]

A difficulty in putting together an environment based on message passing is standardizing the messages so that messages added by new clients do not conflict with existing messages and so that it is clear, when a client is to be integrated, what messages it should look for. A set of conventions should be adopted that would categorize the various messages. Because FIELD is a prototype system, the set of messages has evolved along with the system, and the set of conventions has evolved as well.

The set of messages used by FIELD can be divided into two categories, command messages and informational messages. Command messages are directed toward a specific tool and are generally sent synchronously. Informational messages are broadcast by a tool to all other tools that might be interested in them. These are generally sent asynchronously. We have used different conventions to denote these different classes of messages. Command messages consist of the name of the recipient, followed by a command name, followed by the system name, followed by any arguments to the command. Informational messages consist of the sender's name, the name of the message, the system name and any arguments associated with the message. The sender and recipient names are consistent for a single tool and may be different for a single tool if commands and messages might otherwise get confused. For example, because the debugger has such an extensive set of commands, its recipient identity is DDT, while its sending identity is DEBUG; the cross-referencer, on the other hand, only accepts a small set of commands, and since there is no confusion, uses the same identity, XREF, for both types of messages. The system name allows multiple systems to be debugged at one time. Each message can be directed to a specific system by putting the system name in it; it can be directed to all systems (or any system), by replacing the system field with an asterisk. The current messages used in FIELD are shown in tables 1 and 2.

The annotation editor uses messages in two ways. Its primary use is in connecting annotations to either the source or to other tools. These messages are described for each annotation using a resource file. The particular annotations and their associated messages are described in the next section of this paper. The annotation editor also uses the message server to communicate with the rest of the environment. It accepts two commands, RQST_SAVE to request that the current file be saved and RQST_CLEAR to

Annotation Editor Messages

Commands

ANNOT RQST_SAVE <file>
ANNOT RQST_CLEAR <file> <type>

Information

ANNOT SAVE <file>
ANNOT ABORT <file>
ANNOT EDIT <file>

Make Interface Messages

Commands

BUILD MAKE <system>
BUILD COMPILE <file>
BUILD COMPGO <file>
BUILD COMMAND <system> <command>

Information

BUILD ERROR <file> <line> <text>
BUILD WARNING <file> <line> <text>
BUILD FINISH <status> <transcript file> <command>

Cross-Reference Messages

Commands

XREF QUERY <system> <args> <conditions>
XREF RELOAD <system>

Information

XREF SET <file> <line> [LEFT | MID | RIGHT | NONE]
FLOW SET <file> <line> [LEFT | MID | RIGHT | NONE]

Message Server Messages

Information

MSG FILE_WD <directory>
SERVICE CHECK <service>
SERVICE START <service>

Table 1: Non-debugger FIELD messages

Debugger Messages

Commands

```
DDT ACTION <system> [INIT|QUIT|KILL|STOP]
DDT ASSIGN <system> <var> <expr>
DDT CALL <system> <rtn> <args>
DDT DUMP <system> <from> <to> <length> <format>
DDT EVAL <system> <expr>
DDT EVENT ADD <system> <file> <func> <line> <expr> <cond> <addr> <action> <fgs>
        <action> = TRACE|BREAK|CALL|WATCH|MONITOR|UPDATE|WHEN|
                TRIGGER|STEPSTEP|NEXTNEXT|DISPLAY|<event_name>
        <fgs> = 1:internal, 2:external, 4:event
DDT EVENT SHOW <system> <file> <func> <line> <expr> <cond> <addr> <action> <id>
        <action> = TRACE|BREAK|CALL|EVENT|MONITOR|TRIGGER
DDT EVENT REMOVE <system> <file> <func> <line> <expr> <cond> <addr> <action> <id>
        <action> = TRACE|BREAK|CALL|EVENT|MONITOR|TRIGGER
DDT RUN <system> <args> <in> <out> <new>
DDT SET <system> [INFILE|OUTFILE|USE|WHERE|NEWSYS|DDTOUT|WD] <arg>
          [PRINTIGN|PRINGUSE|CATCH|IGNORE]
          [PROG|ENDPROG] [FORCE_RUN] [RUN_ARGS] [USER_TTY] [ENV]
          [STACK_TOP|STACK_BOTTOM|STACK_DUMP|STACK_SHOW|STOP_UPDATE]
          [SEARCH|BSEARCH|STACK_NO_MAIN|PICKLEVEL]
DDT SHOW <system> [SIGNAL|RUN|USE|SYSTEM|LOCATION|FOCUS|ENV]
DDT STACK <system> <from> <to> <dump>
DDT STEP <system> <count> <sig> [STEP|NEXT|CONT|STEPI|NEXTI]
DDT STOP <system>                                   - NOT queued
DDT SYMINFO <system> [WHICH|WHAT|WHERE|VARINFO|TYPEINFO] <file> <func> <line> <name>
          [LIST_FILES|LIST_FCTS|LIST_VARS|LIST_TYPS]
DDT VIEW <system> <file> <func> <line> <count> <stack_delta>
```

Information

```
DEBUG VALUE <system> <file> <line> <var> <value>
DEBUG ENTER <system> <file> <func> <line> <value>
DEBUG EXIT <system> <file> <func> <line> <value>
DEBUG AT <system> <file> <func> <line>
DEBUG ATSOURCE <system> <file> <func> <line>
DEBUG FOCUS <system> <file> <func> <line>
DEBUG START <system>                                [call]
DEBUG CONTINUE <system>                             [call]
DEBUG STOP <system> <signal_name|OK>                [call]
DEBUG FINISH <system>
DEBUG CLEAR <system>
DEBUG RESET <system>
DEBUG SYSTEM <system>
DEBUG NO SYSTEM <system>
DEBUG NEWSYS <old> <new>
DEBUG EVENT <ADD|REMOVE> <system> <id#> <TRACE|BREAK> <file> <line> <text>
DEBUG WHERE <system> <level> <file> <func> <line> <addr> <args> [A|L|G]
DEBUG WHERE_DUMP <system> <level> <name> <value>
DEBUG WHERE_BEGIN <system>
DEBUG WHERE_END <system> <level>
IE <type> <system> <file> <line> <value> ...        [call | msg]
UPDATE <system> <file> <line>                       [call]
```

Table 2: Debugger FIELD messages

request that all annotations of a given type be cleared for a given file. These allow an external tool to

save a file and remove old error and warning annotations before doing a compilation, for example. It also sends out three informative messages, EDIT to show that a given file is currently being edited, ABORT to show that an editing session was aborted, and SAVE to show that a file was saved. This last message can trigger compilation if desired.

The make interface handles four different command messages. The command MAKE does a make of the given system; the commands COMPILE and COMPGO compile and compile and load respectively for the given source file; and the command COMMAND issues the given command in the context of the given system. This facility also sends out three types of informative messages. The first two, ERROR and WARNING, are sent when the output of the make matches internal error message patterns. They are tied to error and warning annotations respectively. The third, FINISH, is sent out to show that a make has been completed. It it used by the cross referencer to force it to bring its database up to date. It can also be used by the debugger to automatically reinitialize for the rebuilt system.

The command message interface to the debugger provides full access to all debugger commands. This is essential since the textual interface to the debugger maps all user requests into messages and communicates with the debugger through the message interface. The prefix on the command messages can be either DDT or DDTR. The first of these is used by commands that are issued by the user or that should output as part of the user transcript. DDTR messages are used by other tools to get information from the debugger. The two sets of messages are queued by the debugger on separate queues so that other tools messages can be handled at any time (such as while the program is executing) while user messages are only handled when the user is talking to the debugger, and so that other tools can be given priority. The only exception is the STOP message that halts execution and is always handled immediately.

The debugger also sends out a range of informational messages. The VALUE, ENTER, and EXIT messages are sent when handling variable or function tracing. The AT, ATSOURCE, and FOCUS messages show the currently line being executed and the current debugger focus. The START, CONTINUE, STOP and FINISH messages are sent to convey the current execution state of the given system. The

CLEAR, RESET, SYSTEM, NO SYSTEM, and NEWSYS messages convey the current state of the debugger. The EVENT messages show the creation and removal of breakpoints and other related debugger events. The WHERE messages convey the current run time stack. Finally, the messages IE and UPDATE represent special events that can be triggered by the debugger. An IE message generates an interesting event for another tool such as the algorithm animation package. The UPDATE message is sent to tell all tools that they should update their displays. Most of these messages are sent asynchronously. The START, CONTINUE, STOP, UPDATE, and, if requested, IE, messages are sent synchronously to insure that other tools can interact correctly with the debugger.

The cross-referencer handles two command messages. The first, QUERY, allows other tools to issue arbitrary relational queries into the active database. The second, RELOAD, tells the cross-referencer that its database should be reloaded, presumably because something has changed. The cross-reference interface sends out the SET message whenever the user clicks on a listed item. This message is used by the annotation editors to display the corresponding source file and line. Other browsing tools, notably the call graph viewer, send out a similar message when the user clicks on an item.

Finally, the message server itself uses messages to provide higher-level services. It sends out the informational message FILE_WD to show that the current working directory has been changed. This is used, for example, by the make interface to note what makefile should be used. The two SERVICE messages allow the message service to check for and start up services, insuring, for example, that only one cross-referencer exists at a time.

5. Annotations

The annotation editor is used for all interactions between tools in the FIELD environment and the source file. It does this by defining a set of annotations and tying these into the message server. Tools show locations in the source by having an appropriate message trigger an annotation at the proper source location. Tools are invoked based on the source by having an annotation request trigger an appropriate command message. A more complete description of the annotation editor and its design can be found

elsewhere.[11]

Each annotation that is defined can be associated with four different message patterns. The ADD message is sent by the annotation editor when the annotation is requested by the user by clicking on the annotation window. The REMOVE message is sent when the user requests that the corresponding annotation is to be removed. When a message matching the SET pattern is received by the annotation editor, a corresponding annotation is created. When a message matching the UNSET pattern is received, the corresponding annotation is removed.

The message patterns used by the annotation editor consist of strings with embedded formating sequences. These sequences are filled in (for ADD and REMOVE) or interpreted (for SET and UNSET) by the annotation editor to find and document the corresponding annotation. The sequences are shown in table 3. The remaining text must be matched exactly. These are extensions of the basic message patterns used by the FIELD integration mechanism. Annotations can also have different properties. Annotations that are unique can only occur once in any file for a given system. For example, there can only be one line currently being executed. Any existing instances of these will be automatically removed when a request for a new one to be added is made, either by the user or by the system. Moreover, annotations can be made unique with another annotation. All such annotations are considered the same in determining when to automatically remove an annotation. Annotations can be saved as part of the file or just maintained for this run of the environment.

A variety of different annotations are defined in the default version of FIELD. Additional annotations can be defined by users wishing to customize the environment. The annotations connect the annotation editor to the other tools of the environment. Most of the current annotations tie the editor with the debugger. These are shown in table 4. The BREAK annotation can display and request breakpoints; the TRACE annotation can display and request trace points. The WATCH annotation shows variables being traced at a given line. The FOCUS annotation shows the current debugger focus and can request a change in focus. The CURRENT and CURRENTSRC annotations show the current line being executed. The

%F	File name
%L	Line number
%S	System name
%T	First string value associated with annotation
%T1	First string value associated with annotation
%T2	Second string value associated with annotation
%T3	Third string value associated with annotation
%V	Integer value associated with annotation
%d	Ignored integer
%s	Ignored string

Table 3: Escape sequences in annotation message patterns.

UPDATE annotation creates a request to update all views when execution comes to a given line. The EVENT and AEVENT annotations request the placement and generation of interesting events for algorithm animation. The CALLFROM annotation shows the current calling location in the source. Note that not all annotations have all four patterns defined. Annotations that have no ADD or REMOVE pattern can not be explicitly added or removed by the user. Annotations that have no SET or UNSET pattern can only be added or removed respectively by the user.

Other annotations relate the source to other portions of the environment as seen in table 5. Four cross-reference annotations and four call-graph reference annotations are defined, one generic one in each case and specific ones for each button. All eight of these annotations are considered as unique with each other and thus only one can exist at a time. Multiple annotations are defined here so that different buttons can, if desired, have different associated actions. In particular, in the student version of FIELD, clicking with the right button will cause a split display in the editor and will place the corresponding source line in the bottom half; clicking with the left button will place the source line in the upper half of a split display; and clicking with the center button will cause a single window to reappear with the source line selected. The ERROR and WARNING annotations relate compiler error messages, translated into FIELD messages by the make interface, with the corresponding source lines.

BREAK	ADD:	DDTR EVENT ADD %S %F * %L * * 0 BREAK 3
	REMOVE:	DDTR EVENT REMOVE %S %F * 0 * * 0 BREAK %V
	SET:	EVENT ADD %S %V BREAK %F %L %T
	UNSET:	EVENT REMOVE %S %V BREAK %F %L %T
TRACE	ADD:	DDTR EVENT ADD %S %F * %L * * 0 TRACE 3
	REMOVE:	DDTR EVENT REMOVE %S %F * 0 * * 0 TRACE %V
	SET:	EVENT ADD %S %V TRACE %F %L %T
	UNSET:	EVENT REMOVE %S %V TRACE %F %L %T
WATCH	SET:	EVENT ADD %s %V WATCH %F %L %T
	UNSET:	EVENT REMOVE %s %V WATCH %F %L %T
		(UNIQUE)
FOCUS	SET:	DEBUG FOCUS %S %F %s %L
	ADD:	DDTR VIEW %S %F * %L 0 0
		(UNIQUE)
CURRENT	SET:	DEBUG AT %S %F %s %L
	UNSET:	DEBUG CONTINUE %S
CURRENTSRC		*(UNIQUE)*
	SET:	DEBUG ATSOURCE %S %F %s %L
UPDATE	ADD:	DDTR EVENT ADD %S %F * %L * * 0 UPDATE 9
	SET:	EVENT ADD %S %V UPDATE %F %L %T3
	UNSET:	EVENT REMOVE %S %V UPDATE %F %L %T3
		(SAVE)
EVENT	ADD:	DDTR EVENT ADD %S %F * %L %T2 * 0 %T1 13
	REMOVE:	DDTR EVENT REMOVE %S %F * %L * * 0 * %V
	UNSET:	EVENT REMOVE %S %V TRIGGER %F %L %T3
		(SAVE)
AEVENT	ADD:	DDTR EVENT ADD %S %F * %L %T2 * 0 %T1 5
	REMOVE:	DDTR EVENT REMOVE %S %F * %L * * 0 * %V
	UNSET:	EVENT REMOVE %S %V EVENT %F %L %T3
CALLFROM	SET:	WHERE %S 2 %F %s %L 0 L
	UNSET:	WHERE_END %S 3

Table 4: Annotations used with the debugger interface

XREF	SET:	XREF SET %F %L %s
XREF_L	SET:	*(UNIQUE(XREF))* XREF SET %F %L LEFT
XREF_M	SET:	*(UNIQUE(XREF))* XREF SET %F %L MID
XREF_R	SET:	*(UNIQUE(XREF))* XREF SET %F %L RIGHT
FLOW	SET:	*(UNIQUE(XREF))* FLOW SET %F %L %s
FLOW_L	SET:	*(UNIQUE(XREF))* FLOW SET %F %L LEFT
FLOW_M	SET:	*(UNIQUE(XREF))* FLOW SET %F %L MID
FLOW_R	SET:	*(UNIQUE(XREF))* XREF SET %F %L RIGHT
ERROR	SET:	BUILD ERROR %F %L %T
WARNING	SET:	BUILD WARNING %F %L %T

Table 5: Annotations used with other interfaces

Different annotation editors are defined by providing the set of annotations that they display and the set of annotations that they monitor. If an annotation is being monitored by the editor, then the editor will automatically change the file and line that it is displaying so that the corresponding annotation will be visible to the user. The settings for the default annotation editors provided by the environment are shown in table 6. Additional editors can be added as part of user customization of the environment. Moreover, the user can change both sets of annotations interactively while running FIELD.

6. Discussion

The FIELD environment is currently being used at Brown for a variety of applications. The simplified version of the system described above is being used by students in the introductory Pascal course with programs up to 2,000 lines long. FIELD is being used by researchers working with C++ on systems ranging up to 50,000 source lines. Finally, we use FIELD to debug itself and other large

annotview	Display:	BREAK, UPDATE, TRACE, CURRENT, FOCUS, ERROR, WARNING, XREF, FLOW
	Monitor:	XREF, FLOW, ERROR, WARNING
annotedit	Display:	BREAK, UPDATE, TRACE, CURRENT, FOCUS, ERROR, WARNING, XREF, FLOW, EVENT
	Monitor:	
annotddt	Display:	BREAK, UPDATE, TRACE, CURRENT, FOCUS, ERROR, WARNING, XREF, FLOW, EVENT
	Monitor:	CURRENT, FOCUS
aedit	Display:	
	Monitor:	
codeview	Display:	CURRENT
	Monitor:	CURRENT, FOCUS

Table 6: Default annotation editors

(100,000+ line) systems. Because the system is based on existing UNIX tools and because the message traffic is dependent on the user's actions and not on the complexity of the system being debugged, FIELD scales up without serious problems.

In this paper we have attempted to show that the FIELD programming environment provides a high degree of functionality combined with a high degree of tool integration. It does this through a combination of the underlying message-passing integration mechanism, the use of annotations on the source, and a consistent user interface among the tools.

The underlying message facility provides a simple means whereby a broad collection of tools can be easily combined. Developing an external interface for an existing tool involves two steps. First, one develops a mouse and button oriented front end that is consistent with the other tools in the environment. This typically provides buttons to effect the various commands supported by the existing tool. Secondly,

one defines and provides a message interface. This involves first registering patterns with the message server corresponding to those commands that will be offered as a service to other tools. These messages are tied into the command routines that are developed for the workstation interface. The second step here is to generate whatever messages might be relevant to other tools. These generally reflect output gathered from the existing tool or state changes in the interface. Our experience is that we can build an interface to an existing tool in about a week.

Annotations offer several advantages in providing integrated access to the source code. Current environments that provide multiple source displays with different functionalities are confusing and are not user friendly. Annotations can describe locations in the source that are specified by various tools such as the locations of errors found by the compiler or references found by the cross-referencer. Moreover, annotations are a logical means for referencing source positions for other tools, such as for setting break-points in the debugger. This dual functionality is combined in the annotation editor using a flexible approach both to defining new annotations and to defining different instances of the editor that behave in different ways.

For program and data visualization, FIELD makes extensive use of the high-level structured display facilities provided by the GELO package. This is used by the call graph tool to show the call graph, by the make interface to display the dependency graph for a file, and in the data structure display tool. This tool has proven itself flexible enough to readily handle these applications with a minimum of coding within the tool. For example, the call graph display was created in two days without having to write any graphics code.

While the FIELD environment has progressed to the point where it provides a rich set of integrated and useful tools, and while it is currently being used in both instruction and research at Brown, several features that are desirable in a programming environment are still missing and experience has pointed out several weaknesses.

One area of weakness is the system performance. The system requires a substantial amount of memory on a UNIX system, both for itself and for all the other tools it runs, notably the compiler and loader as well as the X11 server. A minimal environment that will insure adequate performance on a Sun workstation, for example, requires 12 megabytes of memory. The working set of FIELD itself is about 2 megabytes. Performance is also a problem in trying to use batch-oriented UNIX tools interactively. Beginners, brought up on interpretive environments, expect compilation to be rapid. UNIX, with its minimum three-process compilers and the need to link and load the compiler output, does not offer appropriate turnaround. Another area where performance is a problem involves getting large amounts of information from the debugger. To support data visualization we need to be able to obtain a complete data structure from the heap. This requires many debugger commands to dump each of the relevant objects and fields, and can cause the updates of complex data structures to take minutes rather than seconds.

We are investigating different approaches to these performance problems. Faster workstations will alleviate some of the problems, but others are inherent to the existing set of tools. The existing compilers are designed to generate efficient code in a batch environment. To provide fast turnaround, these tools have to be modified or replaced. One approach to speeding up compilation turnaround would be to have a fast syntax checker in addition to a full compiler. This would catch many of the immediate problems and eliminate most full compilers. This approach is already used in the PROCASE system. Another approach, used by SABER, is to provide an interpreter. Linking and loading time is also significant in todays systems. One tool we developed independently of FIELD is an incremental loader. This has led to a speed up of this phase of compilation by an order of magnitude. These tools are all useful in their own right, that is, they offer advantages to an existing UNIX environment. FIELD offers a framework where any such tools could be readily incorporated into an integrated environment.

The problem of debugger response is more difficult to manage. Today's debuggers are user-oriented, providing a command set and response level that is geared toward direct user interaction. In an integrated framework such as FIELD, it is often other tools that are using the debugger rather than the

user directly. This requires a faster, more flexible, and more powerful debugging approach. We are investigating a different approach where the debugger provides a full a programming language so that complex queries and actions can be defined as programs to be executed by the debugger. This approach should provide a more flexible and powerful user-level debugger as well, and is being developed independently of the FIELD environment at first. FIELD again provides the mechanisms whereby once this tool is operational, it can be easily made part of the existing environment.

Another area of weakness lies in offering FIELD as an open environment. There are two basic problems here. FIELD makes assumptions that make some tools easier to integrate that others. The two most difficult to incorporate are new debuggers and new editors. The debugger interface currently handles several versions of dbx as well as the Gnu debugger gdb. It takes several weeks of work to write the interfaces necessary for a different debugger. This is probably only an issue if you want to port FIELD to a new system, however. A more serious problem occurs with editors. Everybody has their favorite editor that they want to use, and they do not want to switch. FIELD will allow arbitrary editors to be used outside the system and will preserve saved annotations and the like, but it is not currently possible to use other editors within the system. We are working on defining an interface that will allow arbitrary editors to be used.

A third area of weakness lies in the absolute freedom that FIELD offers for messages. Untyped string-based messages follow the UNIX philosophy of simple representations for interprocess communications. But it means that a new tool developed for FIELD has to understand the messages from all existing tools and that existing tools might have to be modified to understand and react to its messages. We have attempted to develop formatting and naming conventions for messages and to have the relationship between messages and programming actions defined in resource files rather than in the code to somewhat alleviate this problem. If the problem becomes serious in the future, we plan to adapt stronger conventions based on an analysis of the full set of messages we then have or envision and to modify the existing tools appropriately.

Two major features of programming environments are currently missing from FIELD. One of these is version control and system configuration management at a higher level than that provided by UNIX make. Several research groups have or are currently working on tools that fit into this category. We hope to take such a tool when it becomes generally available in the UNIX framework and to integrate it into the FIELD environment. The other missing feature is the ability to connect the source program with documentation or specifications. This can be related to our current work in FIELD in that these can be thought of as annotations to the source. However, these annotations are inherently different from the ones we currently deal with because they are permanent rather than temporal. We are looking into the combination of a hypertext database with our annotation editor as an alternative means for providing this type of long-term relationship and for offering additional functionality to the environment.

Finally, our experiences with FIELD have shown some of the weaknesses in the underlying BWE toolkit. These are most notable in the GELO structured graphics package. While this package does provide a high degree of functionality, it still does not offer the best layouts possible and does not give the user any control over the resultant layouts. We are continually redesigning this package to correct these and other deficiencies.

References

1. P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, "CENTAUR: the system," *SIGPLAN Notices* **24**(2) pp. 14-24 (February 1989).
2. Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas, "An overview of PCTE and PCTE+," *SIGPLAN Notices* **24**(2) pp. 248-257 (February 1989).
3. PROCASE Corporation, "SMARTsystem Technical Overview," PROCASE Corporation (1989).
4. S. I. Feldman, "MAKE: A program for maintaining computer programs," *Software Practice and Experience* **9**(4) pp. 255-265 (1979).
5. S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," *SIGPLAN Notices* **17**(6) pp. 120-126 (June 1982).
6. Sun Microsystems, Inc., *Debugging Tools for the Sun Workstation*. 1986.
7. Robert Munck, Patricia Oberndorf, Erhard Ploedereder, and Richard Thall, "An overview of DOD_STD_1838A (proposed), The common APSE interface Set, Revision A," *SIGPLAN Notices* **24**(2) pp. 235-247 (February 1989).
8. Joseph N. Pato, Steven P. Reiss, and Marc H. Brown, "An environment for workstations," *Proc. IEEE Conf. on Software Tools*, pp. 112-117 (April 1985).
9. Steven P. Reiss, "PECAN: program development systems that support multiple views," *IEEE Trans. Soft. Eng.* **SE-11** pp. 276-284 (March 1985).

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

11. Steven P. Reiss, "On the use of annotations for integrating the source in a program development environment," Brown University (Submitted for publication) (1989).

12. Steven P. Reiss and John T. Stasko, " "The Brown Workstation Environment: A User-Interface Toolkit"," *IFIPS Working Conf on Eng for Human-Computer Interaction*, (August 1989).

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

14. Steven P. Reiss, "Connecting tools using message passing in the FIELD program development environment," IEEE Software (1990 (To appear).).

15. Mark Seiden, "SABER C's edge," *SunExpert* 1(1) pp. 58-67 (November 1989).

16. Richard Stallman, *GNU Emacs Manual*, Free Software Foundation (October 1986).

17. John T. Stasko, "TANGO: A Framework and System for Algorithm Animation," PhD Dissertation (CS-89-30), Department of Computer Science, Brown University (May 89).

18. Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young, "Foundations for the Arcadia environment architecture," *SIGPLAN Notices* 24(2) pp. 1-13 (February 1989).

19. Think Technologies, Inc., *Think's Lightspeed Pascal User's Guide and Reference Manual*. 1986.