On the Use of Annotations for Integrating the Source in a Program Development Environment

Steven P. Reiss

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-32
April 1991

On the Use of Annotations for Integrating the Source in a Program Development Environment†

Steven P. Reiss

Department of Computer Science
Brown University
Providence, RI 02912

February, 1990

Abstract

A central problem in building a program development system is how to integrate the various tools so that the user sees a single view. This integration involves providing a common user interface to the tools, providing a backbone within the program development system that allows the tools to communicate with each other, and providing consistent access to the program for the user. In this paper we examine the use of annotations and an editor that supports annotations as a means for providing common access to the program source for a variety of tools within an open program development system. We show that annotations provide a convenient and consistent way of relating tool output to the source and of initiating commands in the different tools based on the source. We do this by presenting our approach to annotation editing and examining several of the design issues that arose during its development.

Keywords: Editors, programming environments, integration mechanisms, tools.

† 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.

On the Use of Annotations for Integrating the Source in a Program Development Environment†

Steven P. Reiss

1. Introduction

Program development environments provide a variety of tools to assist the programmer in writing, debugging and maintaining programs. Their strengths lie in the richness of their toolsets and in their ability to integrate these tools into a consistent, powerful whole. Integration within a sophisticated set of tools requires more than just having the tools communicate with each other. It requires that the programmer be given a consistent interface, both to the tools and to the program under development. That is, the integration must be done both at the syntactic and at the semantic levels.

Providing a consistent interface to a set of tools is relatively simple. It is accomplished by using a common interface toolset in a consistent manner. Providing a consistent interface to the program is more difficult. A program development environment deals with the program under development in a variety of forms: source code, object code, flow analysis, configuration management, version control, etc. Each of these forms should be accessible to the programmer in a consistent manner. There should be an interface for each form, and all commands and displays relevant to that form should be available from that interface.

The central form used in such an environment, and the most crucial for integration, is the source code. It is important that the programmer be provided with a single tool to access the source, and that this tool be capable of interacting with all the other tools in the environment. In particular, the programmer should be able to edit the source, request that the source be compiled and linked, debug in terms of the source, cross-reference using the source, associate documentation and design information with the source, and relate the source to performance information, all from within a single tool.

The approach we have taken to providing such a tool is to develop an annotation editor that associates arbitrary annotations with lines of the source. The annotations are tied to a message system that the various tools of the environment use to communicate with. Tools can indicate locations in the source using annotations by sending appropriate messages through the message system. These locations and auxiliary information associated with the annotations are displayed with the source in the editor. Moreover, the user can issue commands related to the source by requesting annotations be placed on the file. Requesting a BREAK annotation, for example, causes a message to be sent that results in a breakpoint being set at the appropriate line.

This tool is part of the FIELD programming environment,[6] currently in production use for both teaching and research at Brown Univeristy. This environment is described briefly in the next section. This paper describes the basis of the annotation editor as it is used in FIELD, and offers insights into the design of the editor based on our experiences. The subsequent section provides a overview of the annotation editor. This is followed by discussions of how the annotations are integrated with the message system, of how annotations are made part of the file, and of the user interface provided by the editor.

2. The FIELD Environment

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.

Several 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*.[8] 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 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.[8]

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 FIELD user interface was built using the workstation tools we have been developing over the past seven years in the Brown Workstation Environment.[3,9] These tools offer considerable functionality while greatly simplifying the job of the programmer who has to use them in an interface. 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.[5,7] 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.[10]

An example view of the FIELD environment is shown in figure 1. The window in the upper left is the control panel. It contains icons (currently old English letters) for the various views and windows that

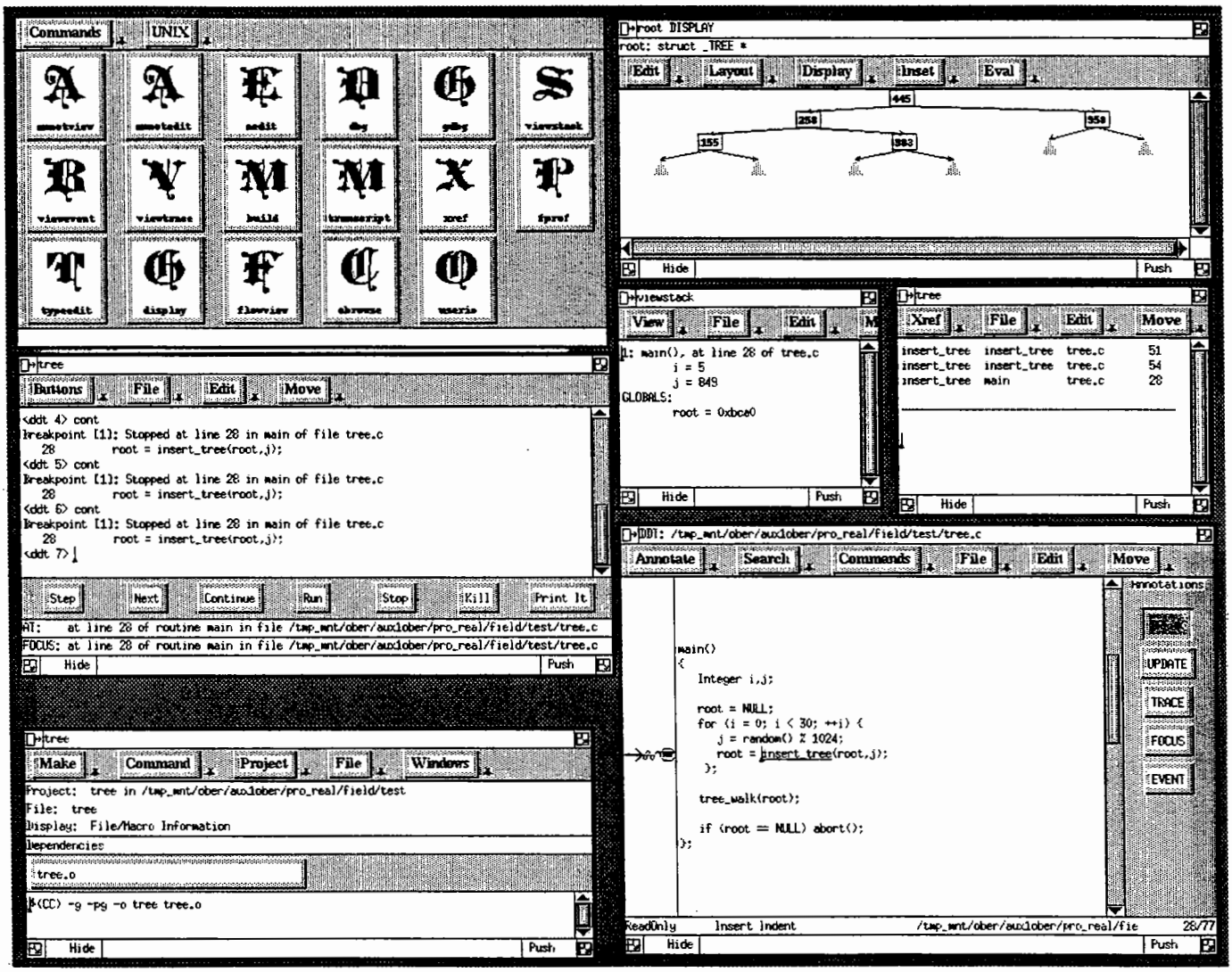

Figure 1: A view of the FIELD environment.

can be defined. Below this is the debugger interface window. The bulk of this window is a transcript of the debugging session. Below this transcript are buttons for debugger commands. In the lower right corner of the display is an annotation editor displaying the source file. There are three annotations

displayed for the one line of text, an arrow indicating the currently executing line, eyeglasses representing the current debugger focus, and a stop sign indicating a breakpoint. The window in the upper right of the display shows a view of the program's data structure. The sample program here does tree insertion, and the tree is displayed in its current state. The dark triangles represent empty subtrees. Below this on the left is a stack viewer. This displays the current function and line being executed as well as the contents of the local variables at this point. The window next to this is a cross-reference viewer. It is displaying the result of a query asking for all calls to the function *insert_tree*. Finally, the window at the bottom left is the make interface. This is currently displaying information about building the system being run.

3. Annotation editing

There are three basic approaches to providing an integrated programming environment. One, exemplified by various implementations of Lisp and Smalltalk, offers a full environment by combining all the tools into a single, closed system. Here all the tools share common data structures and a common address space so that they can easily communicate or invoke each other. It is simple, for example, to click on a name or expression in a Lisp environment and to pass that value to the evaluator or the cross-referencer or the editor. The disadvantages of these systems are that they are complex, that they are closed environments, and that they are not particularly amenable to procedural languages. A second approach, exemplified by UNIX, provides a loose collection of tools that communicate through files in the file system. This has the advantage that the environment is open so that new tools are easy to develop and the environment will work for existing programs. The disadvantage here is that there is little sense of integration for the user. A third approach, the subject of current research, attempts to integrate an open collection of tools using either a database system, or, in the case of FIELD, a message facility. These environments attempt to offer both the integration provided by single system environments and the flexibility of an open set of tools.

One of the central problems in developing an integrated environment based on independent tools is providing a consistent interface to the program source for both the user and for all the tools. In FIELD

this is done by providing a single tool, the annotation editor. This is a full fledged text editor extended with the ability to place and view annotations on any line of any source file.

An annotation on the source file is an external marker that is associated with a particular location of that file. Annotations are not considered parts of the file, but instead provide additional information. In the annotation editor of FIELD, the annotations are used to identify source locations for commands, to serve as markers in the source for other tools, and to define associations between the source and other representations.

The concept of annotations is not new. Its most involved use can be found in various hypertext systems. Here a document is augmented with pointers that can be followed to other portions of the document. Hypertext systems have been used as the basis for software environments.[1] Here they serve to provide a database system for storing the source, documentation, specifications, etc. in such a way that the various items can be easily interrelated. Simpler annotations are also used in programming environments. For example, various visual debuggers such as Sun's dbxtool[2] use annotations to show the location of breakpoints and to the current debugger focus; interpreted systems such as PECAN[4] or, more recently, instruction environments like Lightspeed Pascal[11] use highlighting as an annotations to show the source being executed. These later environments also show the location of error messages directly in the source viewer.

In FIELD we attempt to take the simplified version of annotations used in these various environments and generalize it without adding the full complexity of an underlying database or hypertext system. We first associate functionality with the annotations, for example, a BREAK annotation both represents a breakpoint and can be used to add or remove a breakpoint. Secondly, we provide an open set of annotations, allowing the same annotation mechanism to be used for a wide and extensible variety of applications throughout the environment. Third, we add annotation-based commands to the editor, for example, the ability to search for the next error annotation. Finally, we allow the annotations to be permanently associated with the file, permitting us to use annotations as a simplified form of hypertext pointer.

This is all done in the context of an annotation editor, an example of which is shown in figure 2. This window is divided into five panes. The top contains a menu bar. It contains the Annotation menu with buttons for manipulating annotations, the Commands menu with buttons for issuing generic commands such as compile, and three editor-related menus. The large pane in the center is the actual text editor. The pane on the left is the annotation region. Each line can have an arbitrary set of annotations associated with it here. Moreover, mouse clicks here are used to request new annotations, to obtain more detailed information about existing annotations, and to remove annotations. Finally, the panel on the right lists the types of annotations that the user can request for this window, with the default type, BREAK, currently selected.

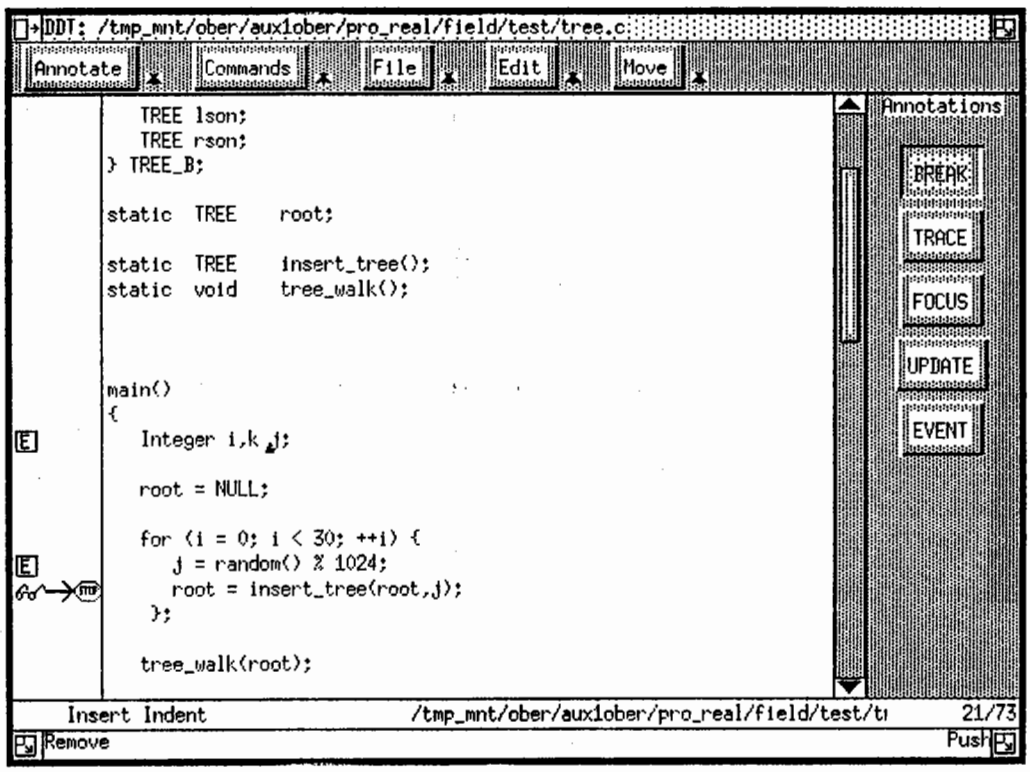

Figure 2: Sample annotation editor

In order for annotations to be an effective mechanism for integrating access to the source file in a programming environment several problems have to be solved. The first problem is tying annotations into the selective broadcasting-based message system that FIELD uses. A second problem is developing appropriate user interfaces for creating, viewing and manipulating annotations. A third problem is developing a scheme for allowing permanent annotations without modifying the source file or restricting the programmer's choice of editors. The next three sections detail our solutions to these problems.

4. Integrating annotations with the message system

Annotations are the means used in FIELD for tying the source to the other tools of the programming environment. As such, they had to be directly tied to the underlying integration framework, the selective broadcasting-based message system that FIELD uses.

The FIELD annotation editor uses a resource manager to allow the definition of arbitrary annotations. Each annotation is defined by providing its name, its iconic description, and a set of associated properties that determine how the annotation behaves. Different types of annotation editors can also be defined using the resource file. Each editor type is characterized by the set of annotations it allows and the set of annotations it monitors. Figure 3 shows a fragment form a FIELD resource file for annotations.

Annotations are tied to the message system in a variety of ways. The most basic way allows messages from other tools to create and remove annotations. Two annotations properties, MSG_SET and MSG_UNSET, are used to define the particular messages that can add and remove an annotation. These properties are string valued, and their values are patterns describing the message that can contain the file name and line number of the associated annotation.

This simple mechanism is extended by additional annotation properties and values. Some types of annotations are unique across the system. For example, there is only one current execution location and one debugger focus. The annotation editors in a FIELD environment must insure that they maintain this uniqueness. This is done by removing any existing annotation of that type when an add message arrives for an annotation that has the UNIQUE property defined. Other annotations contain additional

information. For example, trace events contain the variable to trace while error messages contain the error text. The annotation editor allows three fields of information to be associated with each annotation. These can be filled in from values described in the pattern when the annotation is defined or by the user when requesting an annotation.

Another problem that arises is that of duplicate annotations. Duplicate annotations can arise for a variety of reasons: because there should be duplicate annotations on a line, because other tools send messages that trigger the annotations more than once, or because the editor added the annotation and another

```
ANNOTATION +
  NAME = BREAK
  DISPLAY = "ANNOT_STOP_SIGN"
  MSG_ADD = "DDTR EVENT ADD %S %F * %L * * 0 BREAK 3"
  MSG_SET = "EVENT ADD %S %V BREAK %F %L %T"
  MSG_UNSET = "EVENT REMOVE %S %V BREAK %F %L %T"
  MSG_REMOVE = "DDTR EVENT REMOVE %S %F * 0 * * 0 BREAK %V"
  RESEND = .
  ;

ANNOTATION +
  NAME = CURRENT
  DISPLAY = "ANNOT_ARROW"
  MSG_SET = "DEBUG AT %S %F %s %L"
  UNIQUE = . PRIVATE = .
  COLOR= green
  ;

ANNOTATION +
  NAME = EVENT
  DISPLAY = "ANNOT_EVENT"
  MSG_ADD = "DDTR EVENT ADD %S %F * %L %T2 * 0 %T1 13"
  MSG_UNSET = "EVENT REMOVE %S %V TRIGGER %F %L %T3"
  MSG_REMOVE = "DDTR EVENT REMOVE %S %F * %L * * 0 * %V"
  INFO = ( "Event type" "Parameters" )
  QUERY = . IMMEDIATE = . SAVE = . RESEND = . MULTIPLE = .
  ;

ANNOTATION +
  NAME = ERROR
  DISPLAY = "ANNOT_ERROR"
  MSG_SET = "BUILD ERROR %F %L %T"
  PRIVATE = . ACCUMULATE = .
  COLOR= blue
  ;
```

Figure 3: Sample annotation definitions.

tool sent a message to add it. The first situation can arise from multiple breakpoints being defined on a line or from multiple error messages occurring on a line. The second arises because the status command of the debugger sends an add message for each breakpoint, thereby allowing a new editor to update itself, but confusing existing editors. The third arises when the editor creates the annotation for the user, and another tool that the editor can not assume is running, also sends an add message for that annotation. This can be the case for EVENT annotations, used to denote interesting events for algorithm animation in TANGO,[10] since the editor does not assume that the debugger is running and therefore explicitly creates the annotation.

We have developed strategies for handling each of these cases in the way that seems most appropriate to the user. The default solution is to only allow one annotation on a line. Multiple annotations on a line can either be ignored, as with breakpoints, or can be accumulated as with error messages. The ACCUMULATE property causes the text associated with multiple annotations to be concatenated, allowing, for example, all errors occurring at a given line to be combined into a single annotation whose value contains all the error messages. Where multiple annotations are actually desired, as with EVENTs for algorithm animation, the annotation type can be given the property MULTIPLE. Finally, to avoid the last problem, we don't allow the message system to set annotations that the editor itself adds. This limits the flexibility of the system, but causes the annotation display to be correct.

The other major task for connecting annotations with the FIELD message system is to allow annotations to be used as commands to other tools that refer to the source. To accomplish this, we associate two additional string properties with the annotation type, MSG_ADD and MSG_REMOVE. These are patterns that can be filled in with the current file, the line number of the request, and any of the values associated with the annotation. This allows, for example, a user request for a BREAK annotation at a given line to be translated into the appropriate message to set a breakpoint in that line of the current file. Not all annotations can be set by the user. The PRIVATE property is used to indicate that the corresponding annotation is not directly user definable.

When the user attempts to create a new annotation, the annotation editor can do several things. First of all, the annotation type can have an associated QUERY property, in which case a dialog box is created to allow the user to enter the values that are to be associated with the annotation. Secondly, the annotation may be created by the editor. This is generally *not* the case, however. Most annotations are created as a side effect of the command that is associated with the request. For example, requesting a BREAK annotation sends out the command to define a breakpoint, but the message that the debugger finally sends out when the breakpoint is defined is the one that causes the appropriate annotation to be defined in the editor. However, some annotations, such as EVENT annotations for algorithm animation, can not insure that their associated command will have a side effect and should be created by the editor. This is indicated by the IMMEDIATE property.

A final problem comes up in keeping the source display synchronized with the rest of the system. This is handled in a variety of ways. First of all, each editor has a set of messages that it sends out whenever it opens a new file. These are generally commands that request add messages for any annotations for that file, i.e. requests to the debugger for the current focus and any breakpoints or tracepoints defined in the file to be edited. Secondly, the annotation editor supports a resend capability whereby it can rebroadcast all the add messages for all the annotations of the current file. This is triggered by a message from the message server, and is applied selectively based on annotation type. For example, most of the source viewers will rebroadcast when they receive a message from the debugger indicating that it has reloaded the system (presumably because the object code changed), and will automatically rebroadcast all BREAK and TRACE events.

5. Permanent annotations

In addition to relating annotations to the message system, we need to provide permanent annotations. Permanent annotations are useful for relating the source to other documents such as design, specifications, or documentation. They are also useful in providing the interesting events needed to drive the algorithm animation package, and for allowing graders to provide comments on a student's program.

Our implementation of permanent annotations is complicated by our desire to maintain and emphasize open systems in FIELD. It would have been relatively simple to extend the annotation editor to maintain a set of annotations on a permanent basis, either by keeping information in the file or by keeping a separate file of such annotations. However, we want to allow the programmer to use arbitrary editors and not just the base editor provided with the annotation package.

The first solution we tried was to embed the annotation as a comment in the source file. This worked, but had the drawback that the source file that the user saw in the annotation editor was not the same as that seen in any other editor (or in any other tool that happened to also read the source). This lack of transparency, along with the problems inherent in allowing the user to edit these stylized comments, caused us to search for another solution.

Our current solution is to maintain two additional files for any file that is to contain permanent annotations. One of these files is just a copy of the original file exactly as it was when the annotations were last saved. The other is a list of annotations. In addition, we created another small system that takes these two files and the current version of the source, recomputes the locations of the annotations as needed, updates the saved files, and returns a list of permanent annotations to the annotation editor.

This system works by first running the Unix program *diff* over the saved and the most recent version of the file. The previous list of annotations is read in and, where the corresponding line was not changed or deleted, the line number associated with each annotation is updated. If the line corresponding to the annotation was modified or deleted, then one of two options is allowed. In the first case, the annotation is just moved to the nearest original line of the file. The alternative, is to search the new file to find the line in that file that is closest (in an editing sense) to the original line. This allows the program to move annotations when their attached code is moved.

6. Annotation editor user interface

The annotation editor also presents a number of choices and problems in terms of defining an appropriate user interface. One issue is where to place annotations. The annotations could be put directly

in the file, intermixing annotation icons with the actual text. While this might produce a more consistent interface, and is closer to the interface provided by hypertext and similar systems, it was rejected because it seemed too complex to implement and because we wanted to provide a single consistent editor over the whole system, both for annotation editing, for standard editing, and for straight text input. The approach we take, placing annotations in a separate panel to the left of the text, serves both to make the annotations stand out and to allow a separate interface for creating and manipulating annotations.

This solution is generally acceptable and has been well received. However, there are cases where it is inappropriate. For example, one specialized annotation editor that we have created is a code viewer. It catches messages from the debugger indicating the current line of execution, and moves an arrow icon to indicate control flow through the source, thereby providing a simple form or program animation. In this case, simply moving the arrow icon does not give a good sense of animation. We therefore have an alternative means for displaying an annotation, highlighting the line of the annotation by having it selected within the editor. This simple modification for the specialized editor yielded an informative and helpful animation.

A second issue that arises is what functionality beyond adding and removing annotations is needed by the user. We provide a search facility that allows the user to easily search for the next annotation of a particular type. This is useful for searching for the next error message in the source file. We also provide a rotate command that rotates all the annotations displayed on a given line, allowing more annotations that can fit in the annotation window to be associated and viewable for the line. A query capability for the first annotation of a line is also useful. This allows the user to view a detailed display of information about the annotation, generally including is file position, type, and any associated text strings. It also gives the user the option of removing the annotation or resending any message associated with it.

A third problem is keeping the annotation editor synchronized with other tools. For example, one annotation editor is supposed to keep in step with the debugger, showing the current source code that the debugger is working on. We handle these situations by allowing annotation editors to monitor a set of

annotation types. This means that any message that would normally cause that annotation to be created in a file, will first be analyzed by the annotation editor. The editor checks the file and line number associated with the annotation and changes files and automatically scrolls the display so that that particular line is displayed. Then it adds the annotation. Different annotation editors can be defined monitoring different annotation types. Monitoring has also been augmented to use the editor's capability to split the edit window. Monitored annotations can force such a split and show up in either the top or bottom of the window. This is used, for example, to provide a split view showing the current caller and callee during execution. Finally, an interface is provided through a dialog box allowing the user to dynamically change the set of annotations that a particular editor is monitoring.

Another issue is how to assign annotations to character positions. The usual case is that an annotation is associated with a line of the source. However, situations arise where annotations should be associated with a particular character position in that line. For example, in setting a breakpoint on a line that contains multiple statements, it is desirable to set it at not only the first statement, but also the intermediate statements. Similarly, variable trace events might want to be set on the particular variable being displayed. In the internals of the annotation editor, annotations are actually associated with a file position, i.e. a particular line and character. The difficulty arises in designing an appropriate user interface for this.

Our solution, which is not ideal and needs to be reworked, is to allow the annotation type to specify how it wants to be associated with a character position. When an annotation is requested, if the current file position is on the line of the request, then the annotation can be placed relative to this position, i.e. at the position, at the token indicated by the position, at the next semicolon on the line, at the end of the line, at the start of the line or at the first non-space on the line. The last three of these can also be chosen if the file position is not on the given line. The annotation is still displayed in the left window, but queries about the annotation can describe its actual position.

7. Conclusion

The annotation editor is an integral part of the FIELD environment. Because programming environments are centered around the source program, we have found that integration provided by the annotations is just as important as that provided by the underlying messaging system. Moreover, from the user's point of view, annotations have provided a consistent and extensible interface to the rest of the system.

Annotations have been used for a range of applications within FIELD, ranging from issuing debugger commands to set and remove break and trace points, to viewing error and warning messages, to seeing the result of cross-referencing, to correlating the source with the call graph. Moreover, they have been used to connect the running program to animation display facilities.

Permanent annotations are also being used to correlate different files in the program development environment. One tool that is used for teaching connects the source code with its corresponding pseudo-code, allowing students to visualize execution both in terms of the code and in terms of their design. Similar tools are under development to connect program documentation and help facilities to the source code.

References

1. Norman Delisle and Mayer Schwartz, "A programming environment for CSP," *SIGPLAN Notices* **22**(1) pp. 34-41 (January 1987).
2. Sun Microsystems, Inc., *Debugging Tools for the Sun Workstation*. 1986.
3. 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).
4. Steven P. Reiss, "PECAN: program development systems that support multiple views," *IEEE Trans. Soft. Eng.* **SE-11** pp. 276-284 (March 1985).
5. Steven P. Reiss and Joseph N. Pato, "Displaying program and data structures," *Proc 20th Hawaii Intl Conf System Sciences*, (January 1987).
6. Steven P. Reiss, "Integration mechanisms in the FIELD environment," Brown University (October, 1988).
7. Steven P. Reiss, Scott Meyers, and Carolyn Duby, "Using GELO to visualize software systems," *Proc. UIST '89*, pp. 149-157 (November 1989).
8. Steven P. Reiss, "Connecting tools using message passing in the FIELD environment," *IEEE Software* 7(4) pp. 57-67 (July 1990).
9. Steven P. Reiss and John T. Stasko, "The Brown Workstation Environment: A User-Interface Toolkit"," pp. 215-232 in *Engineering for Human-Computer Interaction*, ed. G. Cockton,North-Holland (1990).

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

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