FIELD Support for C++

Steven P. Reiss and Scott Meyers

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-03
Febuary 1993

FIELD Support for C++[†]

Steven P. Reiss and Scott Meyers

Department of Computer Science
Brown University, Box 1910
Providence, RI 02912.

spr@cs.brown.edu and sdm@cs.brown.edu

Abstract

FIELD is an open, language-independent programming environment offering a wide array of program development tools, including compilers, debuggers, build utilities, cross-referencers, profilers, call-graph displayers, and data structure displayers, as well as a uniform interface to source code via an annotation editor. Its most important tool specifically tailored to C++ development is **cbrowse**, a class browser that we believe to be the most sophisticated available. This paper is devoted primarily to describing the capabilities of cbrowse.

Introduction

FIELD is an open, integrated programming environment built around software development tools running under Unix. Within FIELD, standard tools such as compilers, debuggers, profilers, etc., run as usual, but they are able to communicate with one another by passing messages through the FIELD message server, which categorizes incoming messages and forwards them to only those applications that have registered an interest in receiving messages of that category, a process known as *selective broadcasting*. This mechanism allows a user to, for example, add a breakpoint annotation to a program in FIELD's text editor, which results in the editor sending a message to the debugger informing it where the breakpoint should be set. Similarly, when the debugger arrives at a breakpoint, it sends a message to the editor telling it where it has stopped, allowing the editor to adjust its display to reflect the current location in the program (source file and line). Other publications [Rei90a, Rei90b] describe FIELD and selective broadcasting in detail; interested readers are referred to those publications for further details.

Most of FIELD's tools are independent of the source language, so they are just as useful with C++ as they are with C and Pascal, the other languages currently supported. In addition to the annotation editor, compilers, loaders, and debuggers, such tools include graphical build interfaces (front ends to utilities like **make** and **shape**[KLM+90].), call-graph displayers, data structure displayers, profilers, and cross-reference databases. Language-specific support for C++ consists of a powerful graphical and textual browser, **cbrowse**, as well as enhancements to the debugger interface. This paper describes **cbrowse** and the enhancements to the source-level debugger.

cbrowse

For most people, the first thing that comes to mind when they think of browsers for object-oriented languages is a graphical display of the class hierarchy. For single inheritance,

[†]This paper was originally presented at the 1990 USENIX C++ Conference, April 1990.

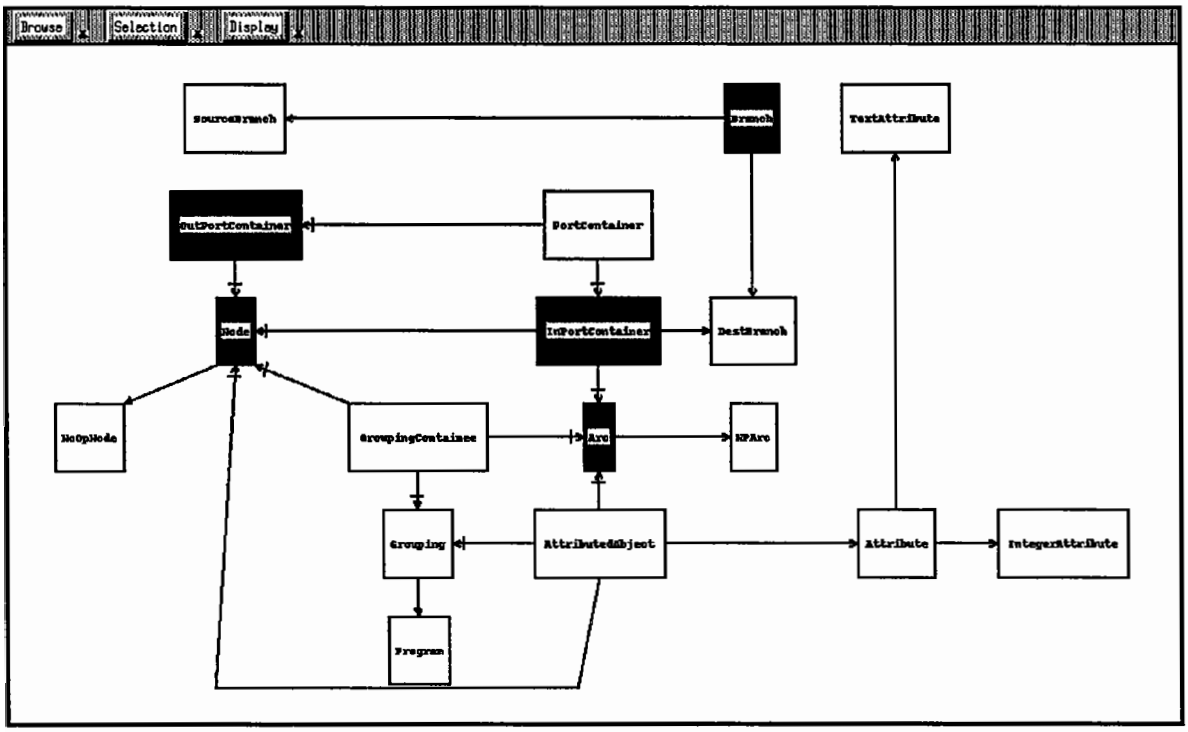

Figure 1: Inheritance Graph Using Rectangular Layout

producing a pleasing graphical display is fairly straightforward, as tree layout is well understood. With multiple inheritance, the problem becomes that of laying out directed acyclic graphs (DAGs), a task that is substantially harder. In general, it is difficult to come up with a single algorithm that will produce aesthetic displays for arbitrary DAGs, so **cbrowse** employs the GELO layout package [RMD89] to offer users a choice of eight different layout algorithms. Figures 1– 3 show how **cbrowse** displays a particular inheritance graph using three different algorithms.[1] Although this graph is more complex than most, with classes having up to four base classes, it is taken directly from a real application. In these figures, shaded boxes contain the names of abstract classes (i.e., those containing pure virtual functions), unshaded boxes contain the names of concrete classes. Arrows run from base classes to derived classes, and virtual inheritance (i.e., inheritance from virtual base classes) is indicated by the presence of a short line behind the head of the arrow. For example, *Node* is a nonvirtual base class of *NoOpNode*, and *OutPortContainer* is a virtual base class of *Node*. In the figures, all inheritance links are public; private inheritance would be indicated by a thinner line style.

In C++ programs, relationships between classes are not limited to those of inheritance. Another important relationship is that of friends, and **cbrowse** can display those as well. Figure 4 shows how **cbrowse** displays a graph showing both inheritance and friend relationships. The dashed arrows in the figure indicate friend relationships, with arrows running from the friend class to the class of which it is a friend.

C++ programs consist of a lot more than classes, of course. There is a lot of information in a class declaration. Its members may be public, protected, or private; may be data or

[1] Practical considerations dictate that each of the bitmaps reproduced in this article be in black and white, but cbrowse, like all FIELD tools, fully supports the use of color in its displays.

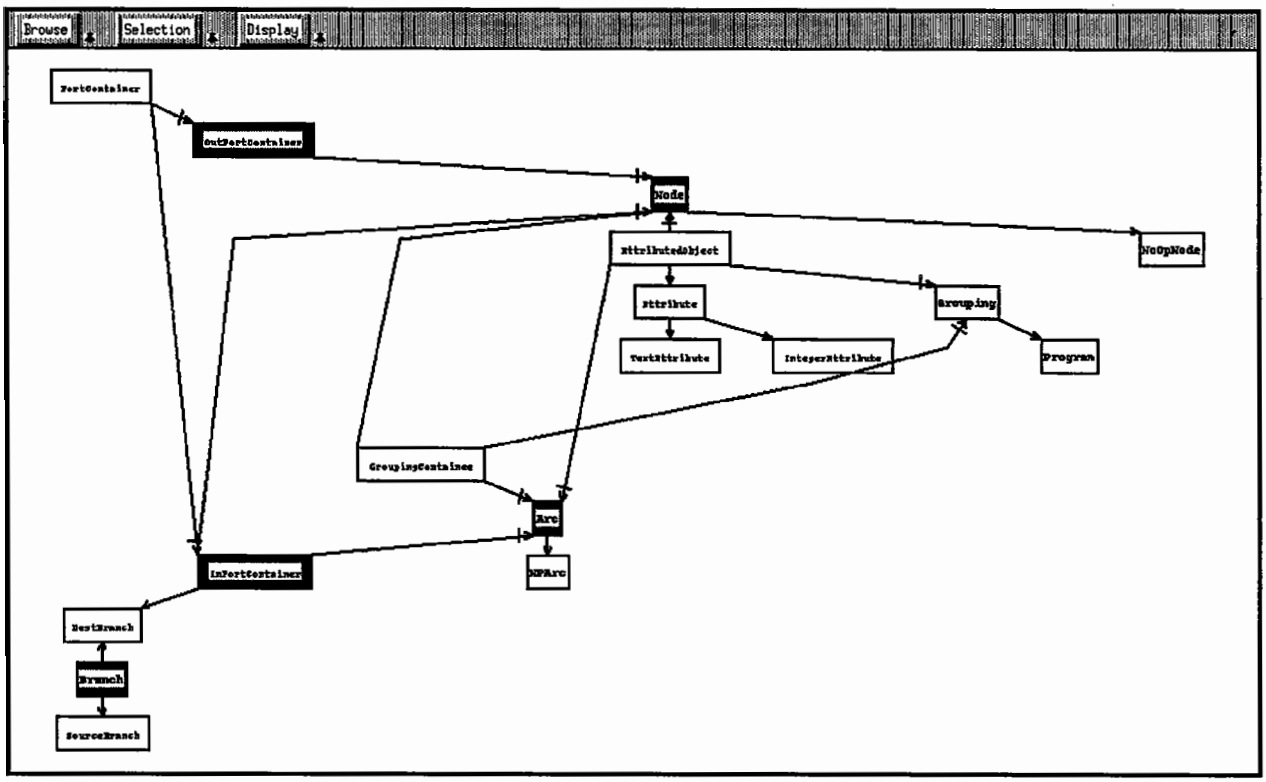

Figure 2: Inheritance Graph Using Grid Layout

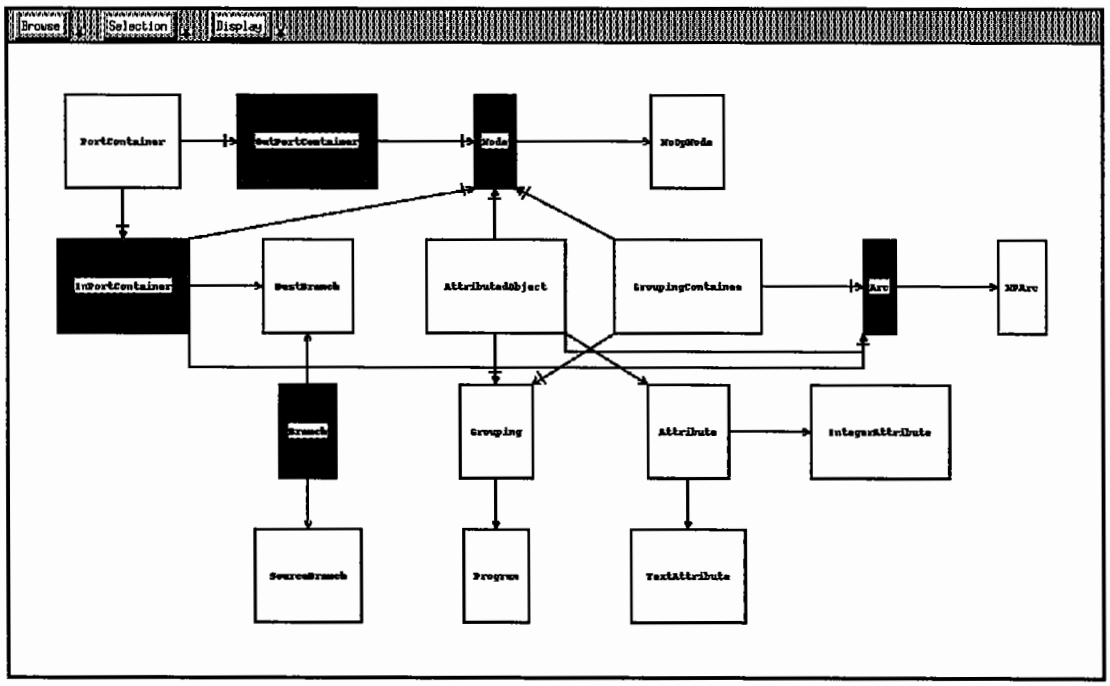

Figure 3: Inheritance Graph Using Breadth-First Layout

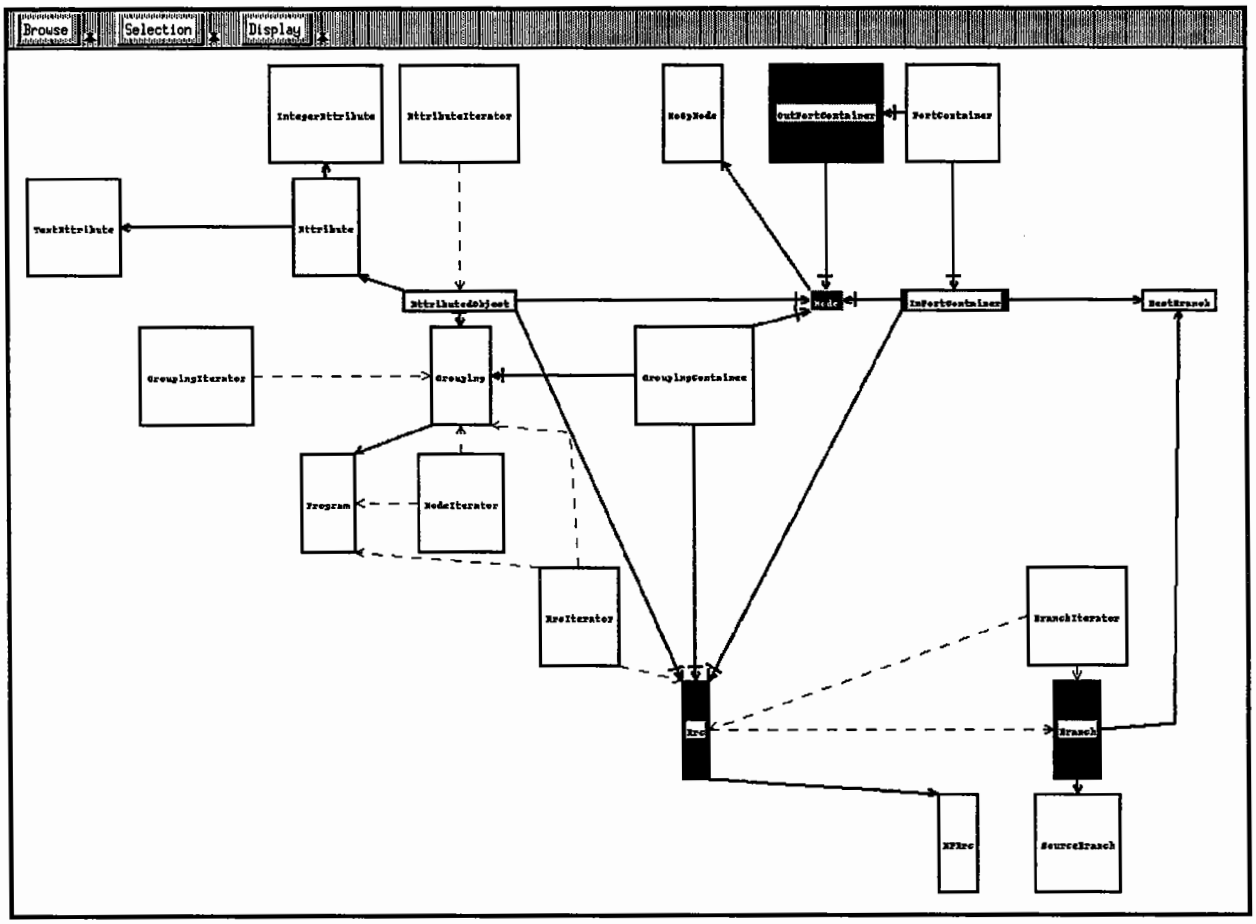

Figure 4: Inheritance and Friend Relationships

functions; may or may not be static; and may or may not be constant. Function members may or may not be inline and may or may not be virtual. If they're virtual, they may or may not be pure. **cbrowse** can display all this information, as shown in figure 5. The figure shows all the information for class *Node*. It is divided into three columns. The leftmost column indicates the accessibility level (private, protected, or public) of each member and differentiates data members from function members. The middle column provides status information for each member (e.g., whether it's constant, whether it's virtual, etc.), and the rightmost column gives member names and indicates whether they are inherited or defined locally. Details:

- **Leftmost column**: Each entry in this column is a triangle, facing either right or left. Right-facing triangles indicate functions, left-facing triangles indicate data members. The fill pattern of the triangle reflects the accessibility of the corresponding members: black is for private members, gray for protected members, and white for public members. Examples: `evaluate_` is a private pure virtual member function; `id` is a protected constant data member.
- **Middle column**: Meanings of status codes are given in table 1.
- **Rightmost column**: Members defined locally are listed as defined. Names of inherited members are preceded by a double-colon. If desired, users can have **cbrowse** also

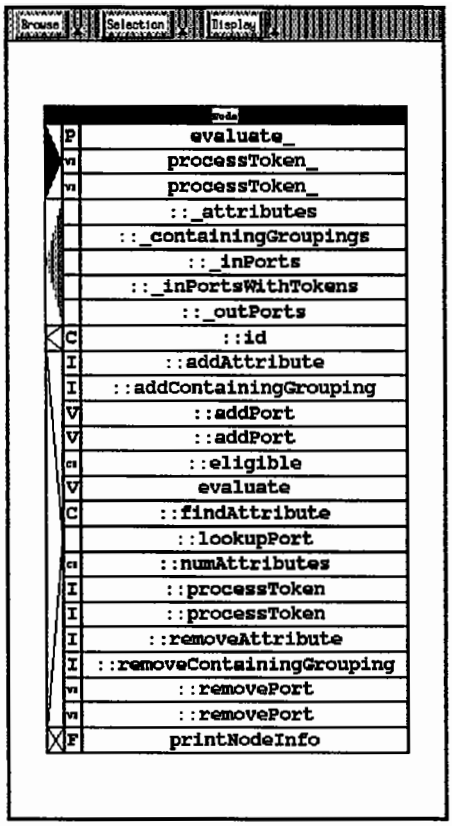

Figure 5: Display of Member Information

show the class from which a member is inherited.

Most of the time, users don't want to see all of this information. For example, potential class users will have little interest in private or protected fields, and potential class developers have no reason to care about private fields. Some users won't care about friend relationships, and others will have no desire to see member properties (such as whether they are constant, static, inline, etc.). **cbrowse** allows users to toggle the visibility of all this information. In particular, users may choose to see or not see data members, function members, private members, protected members, public members, friend links, member access levels and member status data. They may also toggle the visibility of inherited members, thereby displaying either a complete class interface or only those aspects of the interface defined in the class.

Code	Meaning
V	Virtual function
P	Pure virtual function
I	Inline function
F	Friend function
C	`const` member
S	Static member

Table 1: Codes for Member Status Information

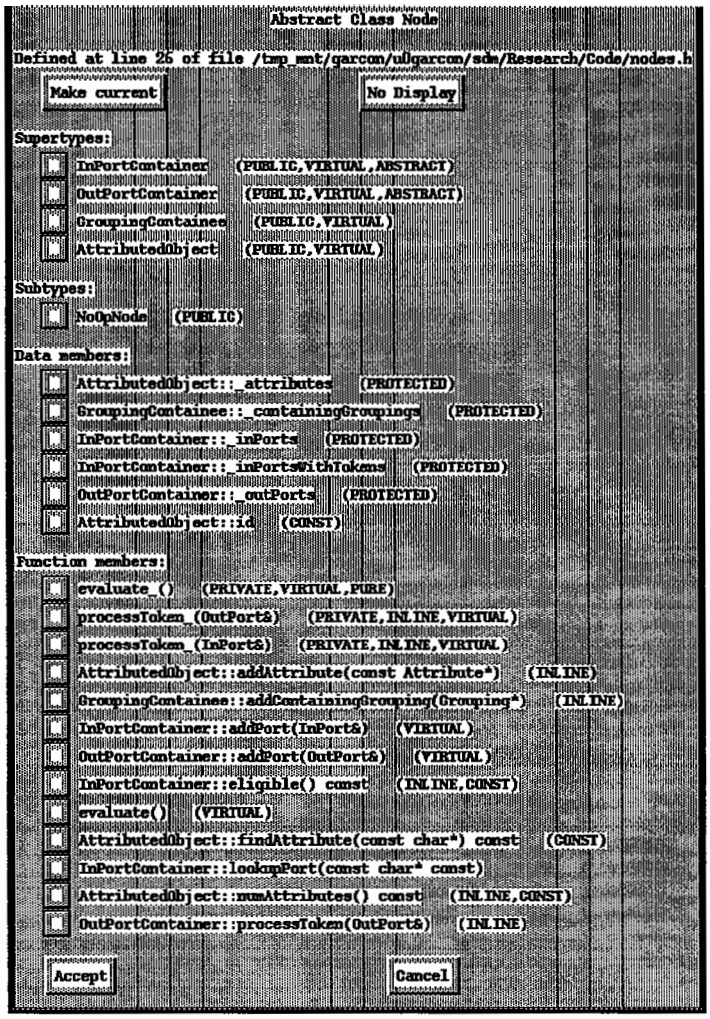

Figure 6: Textual Class Summary

More powerful elision commands are also available. Users may specify classes to include and/or exclude from the display using **grep**-style regular expressions, and may also employ regular expressions to specify which data and/or function members should be displayed. Regular expressions for determining member visibility may be applied on a global basis or to only those classes matching a given regular expression.

Clicking on a member field can bring up a window giving textual information about the member. This information includes the source file and line of the member's declaration, its access level, its properties, and, for functions, its signature. Similar information can be obtained about a class by clicking on the class, as shown in figure 6. It is also possible to open a class information window that will always display a textual summary of the currently selected class. If FIELD's annotation editor is being run at the same time as **cbrowse**, it will automatically bring up the appropriate source file at the line of declaration of the selected class or member.

When inherited members are being shown, selecting an inherited member not only highlights the member, it also highlights the original point of definition, the path by which the member was inherited through the class graph, and the locations "below" the member

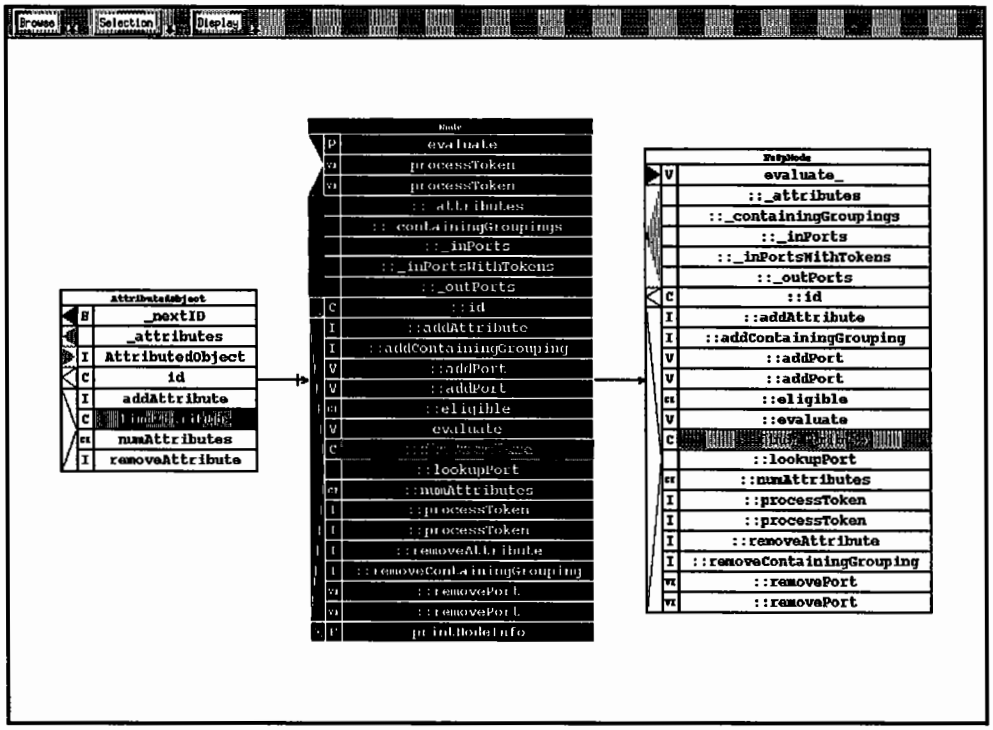

Figure 7: Display Showing Extent of Inherited Member

where the definition continues to be inherited. This facility is shown in figure 7. In the figure, all classes except *AttributedObject*, *Node*, and *NoOpNode* have been removed from the display. The currently selected class is *Node*, hence its depiction in reverse video, and the currently selected member is *FindAttribute*, which is shown highlighted in gray. *FindAttribute* is defined in class *AttributedObject*, so it is highlighted there, too, and the definition continues on "through" *Node* to class *NoOpNode*, so it is also highlighted there.[2] Thus, with a single glance at the graph, a programmer can determine where a particular definition comes from and the extent of its influence. This can be useful in a variety of ways. For example, it can be used to ensure that a virtual function is redefined in all subclasses (if indeed that is supposed to be the case). Alternatively, it can quickly reveal whether a member designed to be inherited has been redefined in subclasses.

During program execution within FIELD, **cbrowse** dynamically highlights the member functions being executed. This allows for a simple form of program visualization.

Other FIELD support for C++

FIELD provides uniform interfaces, both graphical and textual, to the debuggers **dbx** and **gdb**, and performs automatic name mangling on input and name demangling on output for C++ programs. To the usual debugger command set it adds a new command that reveals the

[2]On a color display, the currently selected class and member would be displayed in colors specified for this purpose. There are actually five such colors: one for the currently selected class; one for the currently selected member; one for the original point of definition of the current member; one for members "between" the original definition and the current member; and one for members "beyond" the current selection that share the same definition. Such colors are configurable at initialization by the user.

dynamic type of an object pointed to by a pointer or referenced by a reference. This allows variables to be completely characterized inside the debugger, something that is not possible with standard **dbx** or **gdb**. The FIELD debugger interfaces also allow the specification of breakpoints via regular expressions, e.g., it is possible to break on entry to any function matching a given regular expression. This facility is convenient in its own right, but it is particularly useful for setting breakpoints in virtual functions, which are bound to call sites only at runtime.

All the standard FIELD tools may also be used with C++ , including build utilities, cross-referencers, textual and graphical program profilers, call-graph displayers (with dynamic highlighting), data structure displayers (with dynamic highlighting and modification), and the annotation editor.

Status and Availability

FIELD is a rapidly evolving environment, with new tools being added on a regular basis, and functionality being added to existing tools as users request it. For example, all the C++ support tools have been developed within the past nine months, including **cbrowse**, the new debugging commands, and the necessary parsing and name mangling/demangling software on which they depend. The graphical build utility interface is the most recent addition to FIELD, with the first prototype only recently having become functional.

FIELD itself is based on Unix, the X Window System, and the tools of the Brown Workstation Environment[RS89]. It is currently available on Sun workstations running SunOS 3.x and 4.0.x, and on DEC DECstations and Microvaxes running Ultrix. It may be obtained by sending electronic mail to one of the authors or by requesting a software distribution form from the Brown University Computer Science Department.

Comments and suggestions from users of FIELD and/or BWE are actively solicited. Such feedback frequently results in the addition of valuable new features to the software, often quite quickly after the feedback is received.

Acknowledgments

Support for this research was provided by the NSF under grant DCR 8605567; by DARPA under contract N00014-83-K-0146, ARPA order 6320; and by the Digital Equipment Corporation under agreement 393.

References

[KLM+90] Wilfried Koch, Andreas Lampen, Axel Mahler, Wolfgang Obst, and Uli Pralle. A Tutorial Introduction to the **shape**-Toolkit. Draft from the Technische Universität Berlin, January 1990.

[Rei90a] Steven P. Reiss. Connecting Tools using Message Passing in the FIELD Program Development Environment. *IEEE Software*, pages 57–67, July 1990. Also available as Brown University Computer Science Department Technical Report CS-88-18, "Integration Mechanisms in the FIELD Environment," 1988.

[Rei90b] Steven P. Reiss. Interacting with the FIELD Environment. *Software: Practice and Experience*, 20(S1):89–115, June 1990.

[RMD89] Steven P. Reiss, Scott Meyers, and Carolyn Duby. Using GELO to Visualize Software Systems. In *Proceedings of UIST '89*, October 1989.

[RS89] Steven P. Reiss and John T. Stasko. The Brown Workstation Environment: A User Interface Design Toolkit. In *Preprints of the IFIP WG2.7 Working Conference on Engineering for Human-Computer Interaction*, August 1989.