SDT: A Programming Language for Debugging
(Working Paper)

Steven P. Reiss

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

January, 1989

Abstract

This paper describes the debugger that we are developing as part of the FIELD programming environment. FIELD requires a powerful and flexible debugging framework to facilitate new tools for program and data visualization and for multiple-view programming. Traditional debuggers are too limited to easily fulfill this roll. We are developing a debugger, SDT, that is actually a powerful programming language where the systems being debugged are first class objects. SDT will easily support traditional debugging tasks. Moreover, it should provide the basis necessary for the advanced tools currently under development.

SDT: A Programming Language for Debugging
(Working Paper)

Steven P. Reiss

1. Introduction

We are currently involved in research aimed at producing an integrated, workstation-based programming environment suitable for both software development for research and for teaching. This environment, called FIELD (Friendly Integrated Environment for Learning and Development), has been designed around a simple but powerful integration mechanism that allows a variety of tools, both existing and new, to be easily incorporated. The initial implementation of FIELD is being used at Brown for both software development and in courses. It is also being used as the basis for research into the area of tools for program visualization. Our experiences to date and the requirements imposed by these new tools have led us to the development of a new debugger, SDT.

SDT is more than a standard debugging tool. Most current debugging tools are standalone systems with a fixed set of commands. They are suitable for use as a tool by themselves and provide most of the available functionality to the user. They cannot be easily integrated into a programming environment, nor can they easily adapt to the varied demands that are required by other tools as opposed to users.

SDT is different. It is actually a programming system that implements a complete interpreter for a powerful language designed for controlling and monitoring program execution. This language is based on the non-graphical facilities of Postscript with extensions to facilitate multiple interfaces and multiple systems being debugged, for integrating with the FIELD environment, for accessing and running the system begin debugged, and for obtaining and using information about the symbols of this system. The SDT interpreter is designed to act as a server within the FIELD environment, offering debugging services over one or more systems to the various tools of the environment and talking directly to one or more user interfaces for debugging.

This approach offers both the generality and the flexibility that is required of a debugger in a highly integrated, modern programming environment. Programming environments are sets of integrated tools. Early environments consisted of a fairly standard set of tools:

a compiler, an editor, and a debugger. Today we are seeing more and more tools being developed to fit into the environment, tools such as configuration managers, cross referencers, flow analyzers, testing aids, and performance analyzers. Moreover, new tools are currently being proposed and developed to do dynamic program and data visualization. The key to developing a modern environment that can incorporate new tools is to provide an integration mechanism that allows independent tools to talk to each other and to share information about the program. Within this framework, some tools take on the role of servers, providing information and functionality to other tools. Previous tools required most of this information at the source level. Dynamic visualization tools require information from the running system and therefore must view the debugger as a server within the environment. Current debuggers are not suitable for this role. SDT is designed to be a debugger server that is flexible and powerful enough to provide the functionality necessary for currently envisioned tools and for future tools that might be developed in such an environment.

The debugger server in FIELD is used by a variety of tools within the environment. As one example, the FIELD environment currently contains a data structure display tool. This tool is capable of drawing arbitrary user data structures using either its own standard representations or using easy-to-specify user definitions. Figure 1 shows a tree data structure and its default and user-defined displays. This tool operates by first querying the debugger for type information using the integration mechanisms of the environment. This information is used to define default drawing methods and to allow the user to customize the display. It is also used in the next step where the tool walks through the data structure, querying the debugger to get values for the various elements. To incrementally update its display, the tool must remake all of the value queries.

In most current systems, such a tool would have have to be coded as part of the debugger and not an independent entity. This has the advantage of making the query process efficient and allowing tight

integration. It has the disadvantages of making the debugger overly complex, of making the tool hard to port (since debuggers are hard to port), of inhibiting the further development of this tool or of similar tools, and of limiting the number and type of tools that can be added to the environment.

The programmable framework of SDT allows the best of both worlds. Here the data structure display tool would use SDT as a server to get the type definition for the data structure to display. Then, rather than making numerous queries of the debugger, it could generate a program that would walk through that structure. This program could be used to first dump the initial structure for display and to make a copy of it within the debugger. Later, it could be used to compare the current data structure against the stored copy and to report only the values that are different, allowing efficient update of the display with a minimum of interaction between the debugger and the display tool. The comparison could be done using the lightweight process and event mechanisms of SDT so that it could be triggered whenever the program stopped, whenever it arrived at some set of locations, or every so many milliseconds. The comparison would generate messages to the display tool whenever a change was detected.

In the next section we give a brief description of the FIELD environment with an emphasis on the current and proposed tools that interact with the debugger. Here we illustrate the need for a debugging server that can interact with the tools and point out some of the advanced capabilities that such tools will require from a server. Section 3 attempts to make this more specific by outlining our initial approach to debugging in FIELD, showing where it is weak, and how SDT can be used as a direct substitute. The next section outlines the basic functionality of SDT. Section 5 describes the higher level functionality, essentially how SDT will be used by the various parts of FIELD. We conclude by describing the current status of the system.

2. Overview of the FIELD Environment

FIELD consists of a set of independent tools that communicate through a selective broadcasting integration mechanism.[9] There is a central message server that runs in its own process. Each tool, when it starts up, registers patterns with this server for those messages that it will be interested in. As the tool

runs, it sends messages to the server. These messages are then matched against the stored patterns and are broadcast selectively to all clients that have registered matching patterns. Messages can be either asynchronous or can be synchronous with an associated reply.

This mechanism is simple but powerful. Messages can be used for a wide variety of purposes. They can be used to broadcast general information such as the current line being executed by the debugger. They can be used to request information such as an editor requesting the location of the definition of a function from the cross-reference tool. They can be used as a command interface so that the data display tool can request variable values from the debugger. Moreover, since we have adopted simple conventions to describe the messages, all of which are strings, it is easy to add new tools to an existing system and to reconfigure the system in a variety of ways.

There are two key tools in the FIELD environment. One is the annotation editor that controls all access to the source files. This is a fully-functional, extensible editor with an annotation window along side. Annotations can be associated with each line of the text. These annotations are tied to the message interface so that requesting an annotation can send a message and so that receipt of a message can cause an annotation to be displayed or removed. For example, the user requests a breakpoint by requesting a break annotation on the appropriate source line. This request actually causes a message containing a command to set the breakpoint to be sent to the debugger. When the debugger creates the breakpoint it broadcasts a message indicating that a breakpoint was set. This message is caught by the annotation editor and results in a break annotation being created at the line. This editor allows one source view to be used for editing, building, debugging, cross referencing, profiling, and all other tools that require an interface to the source.

The other tool that is central to the environment is the debugger. While the annotation editor provides access to the source, the debugger provides access to the binary and to the execution of the system. The debugger interface is currently the largest and most complex part of the FIELD environment. The debugger is written as a front end and a separate debugger server. The debugger server accepts com-

mands through the message server from any tool in the environment.

The debugger server is used extensively by the environment tools. Rather than requiring a separate source viewer for debugging, the annotation editor uses the debugger server to allow breakpoint and other debugger events to be defined and removed, will automatically display the current debugger focus, and will highlight the currently executing line. A stack viewer interacts with the debugger to dynamically display the call stack and the contents of local variables. An event viewer displays the current set of debugger events, while a trace viewer displays the current values of all the variables that are explicitly being traced. The data structure display tool mentioned previously provides graphical renderings of user data structures with dynamic updating. A call graph browser interacts with the debugger to dynamically provide a high-level, dynamic view of the control flow within the system.

In addition to its current applications, we are developing a variety of other tools that will need to use the facilities of the debugger. We are working on a tool for defining and working with abstractions of both programs and data. This tool will allow abstractions to be defined after the program has been written independent of the underlying programming language. It will allow all or part of a program to be visualized as an abstraction (such as a state transition diagram), and will allow the programmer to interact with the debugger in terms of the abstraction. The tool will provide similar support for data abstractions, offering graphical representations of the abstractions and supporting programmer interaction in terms of the abstraction. Supporting abstractions in this way requires that sophisticated mappings be performed from actions or events in the actual program to the corresponding actions or events in the abstraction. The debugger must be able to identify the complex conditions that describe the program actions and must be able to act accordingly.

Another application we are developing is an easy-to-program algorithm animation system. Current algorithm animation efforts require that the source program be modified with animation calls and require several days to program a new animation.[2, 4] The system we have been developing, TANGO, uses an algebraic formalism of animations as the basis for a powerful language for describing sophisticated ani-

mations, allowing new animations to be created in the matter of hours.[11] These animations are then tied to programs by means mappings between events that occur during program execution and the events that characterize the animations. Again, it is up to the debugger to correctly define and trap events in the program and to pass these events and corresponding information about variable values and program state to the animation system. Some of the program events that have been desired here are not typically provided by current debuggers. Examples of these are events triggered after a given line has been executed or at the start of the second part of a short-circuit Boolean condition.

A third application we are currently developing is a sophisticated data structure display facility. Most of the previous data structure display packages provided standardized displays of structures and were implemented by implemented by incorporating the graphic display code into the debugger.[1, 7] The facilities currently used in FIELD allow for user-definable display of data structures.[8] Moreover, the data structure display tools are separated from the debugger, using the FIELD message server to obtain information about types and values. The display of a complex data structure can be updated incrementally but only by recomputing the whole display and then differencing the result with the current display. This is both costly and inefficient. In order to do more sophisticated displays, including 3-D views and automatic animation, we require a more efficient mechanism for conveying changes in the data structure from the debugger to the display tool.

Finally, we have been working on providing more intelligent debugging facilities for everyday programming. We feel it is important for the debugger to be able to address the questions that programmers really want answered directly rather than forcing the programmers to get their answers indirectly. The types of questions we have in mind are "How did this variable get this value?" or "How did we get here?" These require a sophisticated and efficient history recording mechanism be tied to the debugger and a database interface be developed to address the history. This is not practical to do with today’s debuggers.

3. The Current FIELD Debugger

Our first attempt to provide such a debugger in the FIELD environment was to wrap a message-based interface around the underlying system debugger. This system was divided into five parts: the textual front end, the graphical front end, the message-based interface, command execution, and the debugger control logic.

The debugger architecture separates the front end from the actual debugger by using the FIELD message server. The textual front end provides a debugging language similar to that of the standard UNIX debugger *dbx*,[5] with additional functionality borrowed from the *gdb* debugger.[10] This front end translates user commands into one or more messages that are sent to the message server. The graphical front end provides a mouse, button, and window-oriented interface that is consistent with the rest of the FIELD environment. The message-based interface connects the debugger with the rest of the environment. The input portion registers with the message server patterns for a set of basic debugging commands. The output portion of this interface is invoked by the debugger to send out informative messages that might be desired by other tools, such as the current focus of the debugger or the contents of the execution stack. The use of this message interface allows tools to interact directly with the debugger. It also allows us to substitute more advanced front ends, or, as we plan to do with SDT, to replace the back end with a more sophisticated one, without affecting the rest of the environment.

The fourth part of the FIELD debugger provides an implementation of basic debugging facilities. Rather than implement a new debugger, our approach has been to use existing source-level debuggers to provide this level of functionality. Thus this is implemented as an interface to *dbx* on Suns and to *gdb* on Microvaxes. This interface translates internal forms of the various debugger commands into appropriate commands for the target debugger. It catches all output from the underlying debugger and translates the results back into internal form for later processing. While this appeared to be a simple and portable solution, it has turned out to be much more complex and trickier than expected. The interface has had to duplicate much of the effort of the system debugger (such as processing the symbol table) in order to pro-

vide functionality that is required by FIELD but is missing in the standard system debugger. Moreover, the differences among the various system debuggers, even the different versions of the same debugger (e.g. *dbx* on Microvaxes, Suns, Apollos), are great enough that they each require a substantial changes to port the code from one machine to another.

The final portion of the debugger involves the logic for processing the internal state and event mechanisms. Essentially, to provide for the type of event processing that other portions of the FIELD environment require from the debugger, we needed to maintain the current state of the debugger and of our front end, and to do a significant amount of processing when a breakpoint or other stop occurs or a message is printed by the debugger. In particular, it was necessary to determine the set of events that are active whenever the program stops executing, to process these events individually by sending appropriate messages or printing to the textual interface, and then deciding whether execution should continue or not.

While this debugger has been successful in providing access to the executing program in the current FIELD environment, it is much more complex than we would like and does not offer the flexibility or power that will be required of future tools. The existing and anticipated tools of the FIELD environment need to have more powerful debugging commands, to define new types of breakpoints and program events, to monitor complex data structures for changes, and to generally have finer control over program execution than is easily achieved by existing system debuggers and thus that is easy to do in the current framework. This is the basis upon which we started to design a new debugger for FIELD.

4. SDT

The type and degree of flexibility we required in a debugging tool called for a substantially different framework than is currently used. Rather than provide a predetermined set of functionality as is done in the current FIELD debugger or in existing debuggers, we decided to implement a minimal level of functionality and a powerful programming language in which arbitrarily complex commands and events can be defined.

This approach is similar to the one that Postscript[3] takes to providing a graphics facility. Postscript is successful because it offers the flexibility to do arbitrary drawing and to allow the drawing commands to be packaged procedurally. The model for our debugger was the Postscript implementation for display embodied in NeWS.[6] Not only was this an interactive system rather than a simple output mechanism, but it also manages multiple processes simultaneously and deals with dynamics. These are characteristics that we also needed in our debugger server.

SDT is implemented as an interpreter for a stack-based language which is essentially Postscript without any graphics (i.e. similar to the Forth language). This is a small but powerful general purpose programming language. We have extended this language both to add functionality for handling interaction, for interfacing to FIELD, and for controlling and accessing the processes being debugged.

4.1. Lightweight processes

The extensions for interaction are needed to allow SDT to function as a server for multiple clients. Here we started with the mechanisms that are used by the NeWS server. These include lightweight processes and a general-purpose event mechanism.

Lightweight processes are used within SDT for a variety of purposes. Each command that comes in from the FIELD message server causes a new process to be invoked. This allows several commands to be executing simultaneously and provides faster response to the various tools with a much simpler control mechanism than the one-at-a-time message processing offered by the current FIELD debugger. Each breakpoint or tracepoint is implemented as a separate lightweight process that waits for the particular event that defines the breakpoint to occur and then takes appropriate action. Each system being debugged has an associated lightweight process that monitors the state of that system, sends appropriate events to other processes, and controls the execution of the system. The use of lightweight processes also allows SDT to act as a server for more than one system being debugged, allowing it to naturally support the debugging of multiple process programs. The operators and semantics of lightweight processes in SDT are based on those defined in NeWS.

4.2. Events

Events are the basic mechanism that lightweight processes use for communication and synchronization. The general event mechanism of NeWS, used in SDT, is well designed for this purpose. It allows a lightweight process to register generic event patterns that describe the events that are of interest to it. Other processes and SDT itself can initiate events which are then sent to all processes with matching interests. These are queued for each process and are processed synchronously within the process. As noted, events are used explicitly by SDT to inform processes about the systems being debugged. They also serve as a wake-up or command request mechanism that the external interfaces use to communicate with the process monitoring the system.

Our event mechanism differs in two ways from that of NeWS. The first is that where NeWS events denote canvases (windows on the screen) and mouse or keyboard events, SDT events denote systems being debugged and addresses within these systems. Thus events can be generated for a particular system or for all systems and for a particular address or all addresses. The second is more complex. In addition to the normal asynchronous event mechanism, we needed a means to insure that all recipients of an event had a chance to fully process it. This is essential for insuring that all action routines have a chance to respond to a program event. We accomplished this by adding an *acknowledge* field to an event and by allowing recipients to specify when they are done processing the event and to return a value. The *acknowledge* field can either be another event or it can be a piece of SDT code. When all the recipients of the event have responded, the *acknowledge* field is examined and either the corresponding event is sent or the corresponding code is executed. In either case, the values returned by all the respondents are available.

This acknowledgement mechanism was designed so that the various debugging actions can be coordinated when the system being debugged stops. The system can stop for any of a number of reasons: a trace messages may need to be generated, variables may have to be examined, breakpoints may be reached, the user could be single stepping until the next line is reached, a program fault, etc. Some of

these require that the program continue, some require that the program single step, and some require that the program stay stopped and the user be informed. The various debugging actions such as breakpoints are implemented in SDT as separate lightweight processes that are waiting on events. The process that is controlling the system gets the first event indicating that the system has stopped. It then sends out a new event to these action processes. Each of the action processes that receives this new event must acknowledge it with an indicator of what the desired action should be (step, continue, or stop). Finally, when all the action processes have responded, the control process gathers the returned action requests and determines whether to restart the system or to report a stop to the user. This scheme for determining the result of a internal stop point has been derived from our experiences with the current FIELD debugger where it has worked well.

4.3. The FIELD interface

The extensions we have added to SDT to interface to the FIELD environment provide integrated access to the FIELD message server. There are four operators, one to send an asynchronous message, one to send a synchronous message, one to reply to a synchronous message, and one to register a message pattern. This last operator takes a string pattern and a code fragment. Whenever the corresponding message occurs in the FIELD environment, the arguments are translated into their SDT form and put into an array, a new lightweight process is started, this array is placed on its stack, and the initial code fragment is executed. This facility is used in SDT to handle all the FIELD debugger command requests. The facility is also used to enable other tools to dynamically establish interactive connections with the SDT server.

4.4. Systems as first class objects

The extensions for controlling and accessing the systems being debugged involve the addition of three new types of objects and their associated operations to the underlying language. The object type *system* is used to represent the program being debugged. The object type *USER_TYPE* is used to represent information about a type from the program. The object type *USER_ADDR* is used to represent information about data of the program. These objects were added as primitive composite objects of the language.

This allowed us to handle type and consistency checking and to provide built-in operators for them.

One *system* object exists for each system being debugged. This object contains all the information about the system including the current state of the system execution, the binary and core file associated with the system, the standard input and output files for execution, the default argument list, and local information for accessing the system. Operations are provided to query and define this information and to initialize a process for debugging. Operators for this object also provide basic control of program execution. This set of operations is based on the UNIX ptrace primitives (run, continue, kill, step, read, write) and on simple modifications to the executing program such as inserting a stop point. This set of operations should be easy to port from one machine to another. Given this set of basic operators, more sophisticated control primitives can be built as SDT programs. In addition to these basic operations, SDT generates events that describe the current state of the system. The two basic events are *systemSTOP* and *systemDONE* indicating that the system being debugged has stopped or terminated respectively.

Associated with each *system* object is a hierarchy of Postscript dictionary objects that provide general access to the symbol table of the program. This hierarchy is arranged by file and function to reflect the scoping of the original program. It is created when the process is initialized for debugging. Providing the complete program symbol table in this form makes it fully accessible to SDT programs using the basic Postscript dictionary operations. A variety of operators to search and find symbols in this table are defined within SDT to facilitate its use. This allows the symbol table and all its associated information to be readily available to both the debugger and to other tools within the FIELD environment. Additionally, sophisticated or special-purpose search routines can be written in SDT to offer run-time cross referencing and query capabilities.

Also associated with each *system* object is a set of address tables. These are internal arrays that can be used to map addresses in the system being debugged back and forth to logical contexts in the source. Three address tables exist, one for files, one for functions, and one for line numbers. The primary operator on these tables takes an address and returns the corresponding context as a file, function, and line

number triple. Other operators exist to find the address of a given line number in a given file and to find the first or last line associated with a given function. This address information was separated from the symbol table because of its size and the specialized manner in which it is used. Three separate tables were used to simplify the needed search routines, enabling binary search to be used to find the current file, function and line without having to scan backwards through the table.

4.5. Other SDT types

USER_TYPE objects represent information about types in the application being debugged. They provide a consistent model of types for a variety of languages, initially C and Pascal. These objects are created automatically as the symbol table is read in for a program. Operations on these objects include querying, formating dumped data, and constructing new user types for debugging purposes.

USER_ADDR objects represent variables in the user's program. They consist of a memory address, either absolute, relative to a frame pointer for a given function, or designating a register in a given function, along with a *USER_TYPE* object that describes the length and interpretation of the data. *USER_ADDR* objects are created automatically for all user variables and are stored in the symbol table associated with the program when a *system* object is initialized. Operations are provided to support address manipulation using these addresses so that symbolic expressions can be evaluated for the given application. Operations are also provided to read and write typed values and to display them in a readable format.

4.6. Expression evaluation

In addition to these basic program control facilities, SDT provides an expression scanner that greatly simplifies the user interface to the debugger. Postscript is a stack-based language where expressions are given in postfix form. Most user interfaces will accept expressions in infix notation and will contain references to variables from the system being debugged rather than references to variables in the interpreter. Rather than force each interface and tool to implement is own parser, SDT provides a sophis-

ticated expression scanner. This scanner translates a string expression into an executable array (code fragment) that can then be executed to obtain the corresponding value. The parser handles both C and Pascal expression syntax. It allows the mixing of variables from the system with variables in the SDT interpreter, generating appropriate operators to retrieve the address or value of program variables.

5. Debugger Control in SDT

The low level interfaces described in the previous section provide the basic functionality that is required from a debugger. SDT provides a higher-level interface built on top of this that provides consistency, better error checking, and additional functionality.

This higher level interface is based on the object-oriented extensions that are again borrowed from NeWS. Postscript provides the facilities required to implement an object-oriented extension. New classes of objects are viewed as dictionaries where the key-value pairs are used to hold structural information, the instance variables, and method definitions. Methods are code fragments that are executed in an environment where the object dictionary is first placed on the dictionary stack and thus its instance variables, etc. are readily accessible. A new operator, *send*, is used to invoke a method in this way.

We have defined two primary classes of objects for SDT. The first, *System*, is used to represent a system in a controlled manner. The second, *SysEvent*, is used to designate an event such as a breakpoint or a trace request. Subclasses of this class are used to describe particular events.

5.1. The class *System*

The class *System* offers a controlled interface to the *system* objects of SDT. The low-level implementation in SDT does not provide any controls on the basic operations on *system* objects, nor does it provide facilities for managing events or multiple command streams. The class *System* is designed to provide these and more. The basic operations on *System* objects are creation, invoking commands, and creating and deleting events. This class provides a monitor process and command queues to insure that commands are executed logically, default standard input and output, and synchronization of events.

Each *System* object includes a lightweight monitor process to control access to the underlying system. When a *System* class object is created, an underlying *system* object is defined and initialized, the various instance variables of the class are initialized, and this monitor process is started. All commands and actions that need to access or modify the system must go through this process. The process keeps track of the state of the underlying system and only issues commands when the system is stopped or terminated. It maintains two queues of commands to be issued. The first queue is for tool-related commands. These can be executed at any time that the system is halted. The second queue is for user-related commands. These have lower priority and are only executed when the user knows that the system is stopped. This dual-queue approach was derived from our experiences with the current FIELD debugger. It allows tools that require dynamic information from the debugger to get the information immediately, while still providing a logical and consistent interface to the user. Commands for both queues can either be synchronous or asynchronous. A request to queue an asynchronous command returns immediately. A request to queue a synchronous command suspends the calling lightweight process until the command has been executed and returns a dictionary containing the string result or an error message.

The monitor process also synchronizes the various events defined for the system using the acknowledge feature of the event mechanism as described in the previous section. When a *systemSTOP* event is received for the underlying system, a new event, *System__STOP*, is sent to any interested events. The replies to this event are gathered along with a default action that is settable within the *System* class object to determine whether the system should be continued or halted.

The *System* class also provides for default standard input and output to the running systems. This is done by creating a pseudo teletype, having operators that send information to this teletype (input from the user), and having a lightweight process that monitors the teletype and handles any output written by the program. This facility is required since SDT is designed to be run as a server process rather than one that directly talks to the user.

5.2. The class *SysEvent*

The class *SysEvent* is used to describe the variety of events. The basic class methods include defining a new event, removing an event, and keeping track of the stop points associated with an event. New events are added by providing a code fragment along with a name and a *System* object. The event is registered with the given system and the code fragment is run in a new lightweight process. If the fragment terminates the event is removed automatically. This is used for specifying temporary events. Otherwise, a remove method will kill the event process and remove its breakpoints.

Actual events are defined within SDT by creating subclasses of the class *SysEvent*. These subclasses take different arguments, generate the appropriate name and code fragment and then use the underlying *SysEvent* methods. For example, the subclass StopAtLine takes two arguments, a file name and a line number, generates a name (Break at line XX of file YY), creates a code fragment that sets a breakpoint at the corresponding address, indicates that it is interested in the event *System__STOP* for the given system at the given address, and then loops waiting for this event and sending an acknowledgement indicating that the program should halt with a breakpoint message when the event occurs.

The combination of the class *System* and the various subtypes of the class *SysEvent* greatly simplify the interfaces that are required to translate messages from the FIELD message server into actions within SDT. They also provide a powerful basis for other tools to implement more sophisticated commands and system events in a consistent manner.

6. Status and Plans

FIELD is currently being used at Brown for both research and teaching. It has proven itself to be a powerful and quite flexible environment in this regard. Its current implementation uses the original FIELD debugger as the support mechanism for the existing set of tools.

SDT is currently under development with a prototype system running in a standalone mode on both Sun3 and Sun4 workstations. We are actively working on making in robust enough and on completing the implementation of the existing functionality of the FIELD debugger so that it can replace it within the

existing environment. Once this is done we plan to add the additional functionality that will be necessary to support the new tools that are currently being developed for the FIELD environment and to support powerful extensions to the debugger that we want to have in a programming environment.

References

1. David B. Baskerville, ‘‘Graphic presentation of data structures in the DBX debugger,’’ UC Berkeley UCB/CSD 86/260 (1985).
2. Marc H. Brown and Robert Sedgewick, ‘‘Techniques for algorithm animation,’’ *IEEE Software* **2**(1) pp. 28-39 (1985).
3. Adobe Systems Incorporated, *Postscript Language Reference Manual*, Addison-Wesley (1985).
4. Ralph L. London and Robert A. Duisberg, ‘‘Animating programs using Smalltalk,’’ *IEEE Computer* **18**(8) pp. 61-71 (August 1985).
5. Sun Microsystems, Inc., *Debugging Tools for the Sun Workstation*. 1986.
6. Sun Microsystems, Inc., *NeWS 1.1 Manual*. 1987.
7. Brad A Myers, ‘‘Incense: a system for displaying data structures,’’ *Computer Graphics* **17**(3) pp. 115-125 (July 1983).
8. Steven P. Reiss and Joseph N. Pato, ‘‘Displaying program and data structures,’’ *Proc 20th Hawaii Intl Conf System Sciences*, (January 1987).
9. Steven P. Reiss, ‘‘Integration mechanisms in the FIELD environment,’’ Brown University (October, 1988).
10. Richard Stallman, *GDB+ Manual*, Free Software Foundation (February 1988).
11. John Stasko, ‘‘The TANGO algorithm animation system,’’ Brown University Computer Science Department Technical Report CS-88-20 (December 1988).

```
typedef struct _TREE *  TREE;

typedef struct _TREE {
      int     value;
      TREE    lson
      TREE    rson
} TREE_B;
```

a) Type definition from source program

b) Default display of a tree

c) User-defined display of the tree

Figure 1: Data structure display facility