A Dataflow Approach to Event-Based Debugging

Ronald A. Olsson
Richard H. Crawford
W. Wilson Ho

Division of Computer Science, University of California, Davis, CA 95616, U.S.A.

SUMMARY

This paper describes a novel approach to event-based debugging. The approach is based on a (coarse-grained) dataflow view of events: a high-level event is recognized when an appropriate combination of lower-level events on which it depends has occurred. Event recognition is controlled using familiar programming language constructs. This approach is more flexible and powerful than current ones. It allows arbitrary debugger language commands to be executed when attempting to form higher-level events. It also allows users to specify event recognition in much the same way that they write programs.

This paper also describes a prototype, Dalek, that employs the dataflow approach for debugging sequential programs. Dalek demonstrates the feasibility and attractiveness of the dataflow approach. One important motivation for this work is that current sequential debugging tools are inadequate. Dalek contributes toward remedying such inadequacies by providing events and a powerful debugging language. Generalizing the dataflow approach so it can aid in the debugging of concurrent programs is under investigation.

KEY WORDS debugging
programmable debugging
source-level debugging
event-based debugging
dataflow
program monitoring

INTRODUCTION

Despite advances in program verification techniques, program debugging is likely to remain an important activity. Unfortunately, it can be frustrating and time-consuming as well. Various tools have been developed to aid the user in debugging activities.* The large majority of recent work in debugging has focused on the debugging of concurrent programs[Baiardi83], [Curtis82], [Elshoff88], [Gait85], [Garcia85], [GarciaMolina84], [Harter85], [LeBlanc85], [LeBlanc87], [Lin88], [Schwartz83], [Smith85], and [Wittie85] and on graphical interfaces for debuggers[Garcia85], [LeBlanc85], [LeBlanc87], [Lin88], and [Adams86]. While those topics are very important, they tend to overlook some fundamental problems of debugging sequential programs, as though those problems had been solved.

This paper takes an opposing viewpoint: sequential debugging remains an area with significant unsolved problems. Specifically, existing sequential debuggers (e.g., *adb*[Maranzano79], *DEBUG*[VMS88], *dbx*[DBX83], *sdb*[Katseff79], and *gdb*[Stallman89]) require the user to interact with the debugger at a very detailed level instead of at the level of the abstractions present in the user's program. Unfortunately, the user has no means, other than mentally or using pencil and paper, of correlating the occurrence of several logically related breakpoints into a single, more abstract occurrence. This capability is desirable since abstractions in an application program often require a series of procedure calls. Another inadequacy of many current debuggers is that they provide only very limited control over the actions taken when a breakpoint occurs. More powerful means of control are desirable, e.g., a loop to print out the contents of a linked list, or a loop to single step execution of the inferior process until the value of one of its variables is greater than another. This paper offers a concrete methodology and practical tools for solving these problems. It presents a new approach to event-based debugging and describes how that approach has been incorporated into a sequential debugger.

The notion of event-based debugging has received much attention, especially for debugging concurrent programs [Elshoff88], [Lin88], [Bates82], [Bates83], and [Bates88]. It offers a way to filter through the large amount of information associated with the execution of the inferior process. An *event* corresponds to the occurrence of some interesting activity in the execution of the inferior process. Generally, a number of low-level events are pre-defined in an event-based system; typical events include control reaching some statement, procedure entry, and procedure exit or, for concurrent programs, the receipt or sending of a message. One important aspect of any event-based system is how lower-level events are combined into higher-level events. Most current event-based schemes

* The term 'user' refers to a person developing and debugging a program. The term 'debugger' refers to a program that aids the user in debugging. The debugger controls the execution of the user's program, which typically executes as a separate, 'inferior' process.

use some form of pattern matching for such recognition.

This paper presents a novel approach to event-based debugging. The approach is based on a (coarse-grained) dataflow view of events: a high-level event is recognized when an appropriate combination of lower-level events on which it *depends* has occurred. Event recognition is controlled by familiar programming language constructs, such as conditional and repetitive statements and user-defined functions. This dataflow approach, therefore, permits "active" event recognition. It allows arbitrary actions, specified in the debugger language, to occur when attempting to form higher-level events. It also permits actions to occur on "near misses", e.g., when some but not all of the desired lower-level events have occurred or when information gleaned from occurrences of lower-level events "nearly" satisfies a desired condition. This dataflow approach is, therefore, more flexible and powerful than existing pattern matching schemes for event recognition. It is also more accessible to the user: the way that users specify event recognition is close to the way that users write programs.

To demonstrate the feasibility of the dataflow approach to event-based debugging, a prototypical debugger, Dalek*, has been developed for C programs running on top of the UNIX operating system. Besides employing this dataflow approach for event recognition, it provides a powerful, expressive debugging language. In particular, Dalek provides the control structures (conditionals, loops, and user-defined functions) that one finds in conventional programming languages. Such control structures are useful to display the contents of an entire linked list (using a while loop) or of an entire tree (using a recursive procedure), and to control the actions the debugger takes in interacting with the inferior process. An example of the latter is to execute the inferior process, without requiring user interaction, until it reaches a particular state of execution that is of interest in debugging. Dalek also provides a rerun mechanism for playing back execution of a program, and a rich collection of pre-defined functions.

The following sections present an overview of the dataflow approach to event recognition and describe how it has been incorporated into Dalek. Several examples are then presented that illustrate Dalek's usefulness and power. Comparisons are made with other debugging research, and the status and future direction of this work are summarized. The results indicate that the problems of sequential debugging can benefit from additional work, of which this effort represents one step, and that the dataflow approach to event-based debugging is attractive. Additional design and implementation issues, and examples appear in References [Crawford90] and [Olsson90].

---

* A Dalek is a semi-mechanical alien life form that was opposed by Doctor Who in its efforts to "exterminate" everything in sight [Tulloch83].

AN OVERVIEW OF DALEK

Background

The reader is assumed to be somewhat familiar with a standard sequential source-level debugger such as *DEBUG*[VMS88], *dbx*[DBX83], *sdb*[Katseff79], or *gdb*[Stallman89]. Typical capabilities of such debuggers include: setting breakpoints at specified function entries, source line numbers, and relative addresses; executing an associated block of commands immediately after suspending execution of the inferior process at a breakpoint; single stepping execution through the program at the source statement or machine code level; displaying values of program variables and memory locations; and displaying sections of source code. These source-level debuggers differ in the languages with which users control their behavior and in the resultant functionality they provide. Two such differences are noteworthy. First, *DEBUG* and *dbx* provide a mechanism for tracing the value of a variable, displaying its value or suspending execution when it changes. *Gdb* and *sdb* do not provide such tracing. Second, *DEBUG* and *gdb* provide dynamically-typed *convenience variables* for the programmer to use within the debugger to compute and store values. In *gdb*, convenience variables are global; in *DEBUG*, they are global or local to debugging procedures. Neither *dbx* nor *sdb* provides convenience variables.

Dalek is based on *gdb*[*]. It enhances *gdb*'s functionality in two significant ways. Dalek extends the original *gdb* language with a conditional statement (not to be confused with *gdb*'s conditional breakpoint command), a looping construct, blocks, local convenience variables, procedures, and functions. It also provides events, an abstraction mechanism used to construct models of the inferior process's behavior (either its intended or undesirable forms) and to monitor the program's actual behavior in terms of the specified forms.

The Dalek Language

A widely held debugging precept is that the activity of debugging should not require alterations to the source code. Yet quite often during debugging, the user thinks of a few simple procedures that would be extremely helpful, e.g., a routine to traverse and print selected elements of a linked list. Injecting such debugging code into the source code, perhaps by conditional compilation, is not an attractive solution. Often, the user has arduously brought the inferior process to a critical point in its execution under the debugger's control before discovering the need for such a routine. Furthermore, seemingly minor modifications to the source code can substantially alter the symptoms exhibited by bugs, especially if pointers are involved.

---

* *gdb* is the debugger developed by the GNU project of the Free Software Foundation.

The debugger should, therefore, provide the user with the ability to write such debugging code. It should also allow the user to direct when the code is to be executed, e.g., from the debugger's command level or automatically at a breakpoint. Furthermore, the debugger should provide a language that contains all the mechanisms that programmers generally find useful. Dalek's language, therefore, provides assignment statements, **`if`** and **`while`** statements, blocks, local and global convenience variables, procedures, and functions.* Dalek code can be written when the user finds it necessary, e.g., when execution of the inferior process is suspended at a breakpoint. The Dalek code and variables reside within the debugger's address space. Thus, storage of these objects does not affect and is not affected by the inferior process.

Events

An event in Dalek is anything of interest that can be specified using procedural methods. Any number of dynamically-typed attributes can be associated with each event. The definition of a *primitive* event specifies a list of its attributes' names and a command block written in the Dalek language. The definition of a *high-level* event is similar to that of a primitive event but it also specifies a list of *constituents*—lower-level events whose attributes it can access and upon which its activation may depend. Event definitions can be given interactively or read from a file.

Event Recognition by Dataflow Methods

In most models of dataflow, tokens flow between the nodes of a directed graph. In Dalek, the leaves of this graph represent primitive events (independent sources of tokens) and the interior nodes represent higher-level events. Initially, activity within the inferior process causes it to suspend (e.g., at a breakpoint), transferring control to the debugger. While the inferior program remains suspended, the debugger triggers the primitive event associated with the activity in the inferior; the debugger generally transfers control back to the inferior process when the computation in the dataflow graph reaches a quiescent state, as described in more detail later. If the primitive event is recognized, it generates a token. Since a token corresponds to an *instance* of an event, a token is a composite

---

* This paper uses several typewriter fonts to help convey the meaning of identifiers: Dalek commands appear in **`boldface`**; Dalek convenience variable and debugger function names appear in `roman`; and inferior process variable and function names appear in *`italics`*.

Dalek inherits many of its commands and much of its syntax from *gdb*. The following are important for this paper. The **`silent`** command suppresses output announcing that a breakpoint has occurred. The **`cont`** command resumes execution of the inferior process. The **`step`** command executes the next statement of the inferior process. Names of convenience variables and debugger functions start with a '`$`'. Comments begin with a '`#`'. A '`\`' indicates that the current line is continued on the next.

object; the information required to characterize a particular occurrence of an event is carried with it as its attributes. Recognition of a primitive event causes copies of a token characterizing that event occurrence to flow along dedicated arcs to all high-level events that depend on it.

In the dataflow model, computation is performed at the nodes. In Dalek, the command blocks associated with events correspond to nodes. Incoming arcs are represented as queues to which arriving tokens append themselves.

Fig. 1 illustrates the dataflow approach to event recognition. A, B, and C represent activities (e.g., breakpoints) within the inferior process whose occurrence can trigger primitive events D, E, and F, respectively, as indicated by the dashed arrows. In turn, the occurrence of those primitive events can trigger higher-level events, as indicated by the solid arrows. For example, the occurrence of D can trigger event G. Note that the occurrence of E can trigger two events, G and H. The two highest-level events, I and J, can be triggered by G and H, and F and H, respectively.

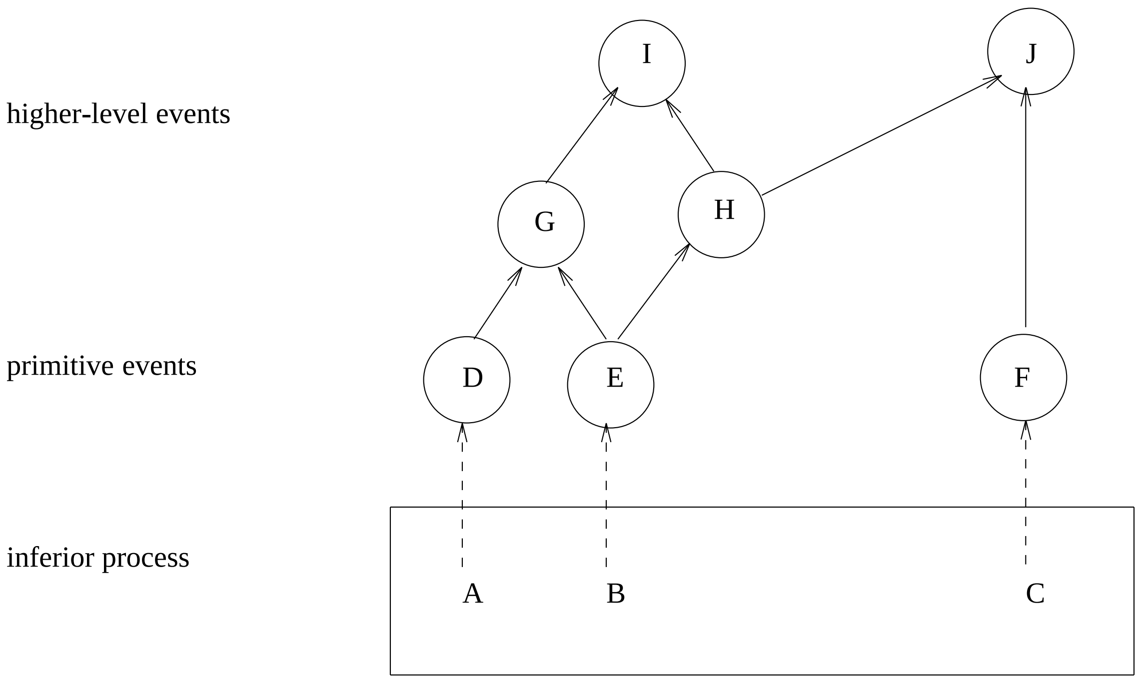

Fig. 1. Event recognition by the dataflow approach.

Primitive Events

In order for an instance of a primitive event to occur, it must first be explicitly raised. Typically, the user specifies an **`event raise`** command in the command block of a breakpoint inserted in the inferior process. Fig. 2 presents a simple example of how a primitive event is defined and raised. `‘element` is defined to have three attributes, `‘v`, `‘nextv`, and `‘kind`; its command block is not shown. Note that event names and attributes are prefixed with a backquote to distinguish them from convenience variables and inferior process variables. Fig. 2 also shows a breakpoint set at line 27 of the source file `foo.c`. Dalek suspends execution of the inferior process when it reaches that line; it then executes the command block. The command block raises the primitive event `‘element` in one of two ways depending on the values of variables in the inferior process.

If desired, a given primitive event may be raised from several different breakpoints in the source code. At the other extreme, a primitive event need not be bound to any breakpoint in the inferior process, as the example in Fig. 3 illustrates. This example involves tracing the value of a global variable. The `‘x_modified` event prints out the

---

```
# Definition of primitive event ‘element; its attributes are ‘v, ‘nextv, and ‘kind.
event define ‘element(‘v, ‘nextv, ‘kind)
   # The command block for primitive event ‘element
     ...
end

break foo.c : 27
commands
   silent # Suppress the default announcement of encountering a breakpoint
   # Raise the ‘element event with one of two sets of attributes
   if (nodeptr→next)
      event raise ‘element (nodeptr→value, nodeptr→next→value,\
                               "middle")
   else
      event raise ‘element (nodeptr→value, "none", "tail")
   endif
   cont # Continue execution of the inferior process
end
```

Fig. 2. Outline of defining and raising a primitive event.

```
# 'x_modified is raised whenever x changes in value.
# Its attributes are x's old and new values and the names of
# the routine in which x changed and that routine's caller.
event define 'x_modified ('oldval, 'newval, 'called, 'caller)
    # Examine attributes of this occurrence of 'x_modified
    if ( $get-attribute ('oldval)  >  $get-attribute ('newval)  )
      printf \
        "x not monotonically increasing in %s called from %s\n", \
        $get-attribute('called), $get-attribute('caller)
    endif
end

# Single step the inferior; raise 'x_modified whenever x changes value
set $old_x_val = x
while (1)
   if ($old_x_val != x)
       event raise 'x_modified ($old_x_val, x, $func(0), $func(1))
       set $old_x_val = x
   endif
   step
endwhile
```

Fig. 3. Tracing a global variable.

names of the procedure and its invoker in which the value of variable `x` is discovered to be not monotonically increasing. It uses `$get-attribute()` to read the values of its own attributes. The **`while`** loop, specified at Dalek's command level in Fig. 3, single steps execution of the inferior process, raising `'x_modified` whenever it discovers a change in the value of `x`. The values of the attributes passed to `'x_modified` are taken from the global convenience variable `$old_x_val`, the inferior process's variable `x`, and the Dalek pre-defined function `$func()`. `$func(N)` returns the name of the procedure `N` frames up the inferior process's call stack. Note that the above example could be more easily programmed without using an event. However, using an event allows subsequent examination of the history of event instances, and correlation of these event instances with others through higher-level events, which can aid greatly in debugging a program.

Primitive events can be thought of as characterizing interesting changes in the control and data state of the inferior process. Typical primitive events include entry to or exit from procedures or smaller blocks of code, as well as changes in a program's data ranging from a change in a variable's value to more complex changes such as a linked list becoming empty, an array becoming unordered, or a graph becoming cyclic. Such complex changes, though, are most often expressed as high-level events.

High-Level Events

The definition of a high-level event specifies which constituent events will trigger that high-level event. That is, the definition indicates which lower-level events' tokens, after arriving on incoming arcs, will activate the event's command block so it can attempt to recognize an instance of that high-level event. A minus sign prefixing a constituent name in the definition of the high-level event indicates that the arrival of a token from that constituent should *not* trigger the high-level event; the token will still be enqueued so that its attributes are available to a future activation of the high-level event.

When a high-level event is triggered, its associated code decides how it should react based on what constituents' tokens are present and the values of their attributes. The code might decide that the present constituents (or some subset thereof) comprise a valid instance of the high-level event. In such a case, to characterize this particular occurrence of the high-level event, the code should assign values to its own attributes—either by copying those of selected constituents or by computing some function based on the available data. Other possible actions include printing a message, reestablishing interactive control while keeping the inferior process suspended, and even modifying data or control flow in the inferior process. The code will then generally remove the selected tokens from its incoming queues so as to avoid repeated recognition of the same tokens when it is subsequently triggered. When the event's code finishes execution, tokens embodying the new event propagate to all high-level events that depend on it, and the recognition phase begins anew. On the other hand, the code might decide that the appropriate combination of constituents is not present on the incoming queues or that their attributes fail to satisfy the desired relationships. To suppress (the normal) generation of tokens representing this event and the subsequent passing of those tokens to the higher-level events that depend on it, the code can execute an **`event dont-propagate`** command.

Transfer of Control

Recall that control transfers from the inferior process to the debugger—e.g., at a breakpoint—allowing the debugger to raise a primitive event. Generally, the code associated with an event executes in its entirety before another event's command block is executed. If a single event triggers multiple higher-level events, the order in which they execute is determined only by the implementation.* When all triggered nodes have executed—i.e., the dataflow graph reaches a quiescent state—Dalek continues execution with the command following the one that raised the initial primitive event.

Exceptions to the above general rule arise when a node executes a command that resumes the inferior process, although such usage is risky and not frequently needed. For example, execution of a single step command suspends the event's code and returns control to the inferior process. The suspended event's code is then resumed after the inferior process executes one statement. Note that resumed execution of the inferior process might encounter a breakpoint, which can cause another primitive event to be raised. Dalek handles such events by saving and later restoring their execution states; the details are beyond the scope of this paper (see Reference [Crawford90]).

Modeling and Exploring Program Behavior with Events

Since each instance of an event has its own attributes (located in Dalek's address space), and since each invocation of an event's command block can have local variables as well, the commands for each event can be written in isolation from those for other events. This structure allows events to mirror the block structure of an inferior process quite naturally. It also permits stepwise refinement of a behavioral model. Thus, the user can define events covering only those abstractions in the program that either highlight a bug's symptoms or that are suspected of causally contributing to bugs.

A history list records all recognized event occurrences. It may be browsed selectively by the user in interactive mode or accessed programmatically via the Dalek language using pre-defined functions provided expressly for that purpose. The event history list allows program behavior to be explored during or after program execution. One such use is described below as part of the examples.

EXAMPLES

The four examples presented below illustrate how the various Dalek features described above combine to form a powerful debugging tool.

---

* The blocks of code associated with the high-level events can depend on the relative order in which they execute. Such code, for example, could depend on a global convenience variable or on the event history list (described later). In general, however, such dependencies are considered bad debugging style. Instead, all dependencies between events should be reflected in the topological ordering of the dataflow graph.

Stack

This example shows how events can be used to monitor the operation of a stack with the usual push and pop operations. In the inferior process, the stack is implemented by an array called *stack* with the variable *stack_pointer* recording the offset of the top of stack from the beginning of the array. Fig. 4 shows the definitions of primitive events `‘push` and `‘pop`, and of the higher-level event `‘push_pop`. When either `‘push` or `‘pop` is activated, the current value of *stack_pointer* is recorded in its attribute `‘stack_p` (using the **event set-attribute** command), which is then passed to `‘push_pop`. In addition to its own name and no attributes, the definition of `‘push_pop` names two constituents, `‘push` and `‘pop`, the lower-level events on

```
# This event is raised when push entered; it records stack_pointer as its attribute
event define ‘push (‘stack_p)
    event set-attribute ‘stack_p stack_pointer
end

# This event is raised when pop entered; like ‘push, it records stack_pointer as its attribute
event define ‘pop (‘stack_p)
    event set-attribute ‘stack_p stack_pointer
end

# Correlate a ‘push with a ‘pop. This event is triggered only by arrivals of ‘pop.
# Its queue for constituent event ‘push stacks yet-unmatched ‘push’s.
# When ‘push_pop executes, its queue for constituent event ‘pop contains only the most recent ‘pop.
event define ‘push_pop ()   −‘push  ‘pop
    # Ensure stack pointer was same for push and corresponding pop
    if ($value(‘push, 0, ‘stack_p) != $value(‘pop, 1, ‘stack_p))
        printf \
          "Stack pointer error: pushed at %d, popped at %d\n", \
          $value(‘push, 0, ‘stack_p), $value(‘pop, 1, ‘stack_p)
    else
        printf "Pushed and popped %d, stack_pointer = %d\n", \
             stack[stack_pointer], $value(‘push, 0, ‘stack_p)
    endif
    event remove ‘push, 0   # Discard instances of ‘push
    event remove ‘pop, 1    # and ‘pop from this event’s queues
end
```

Fig. 4. Events to monitor stack activity.

which it depends. Recall that the arrival of a token from a constituent event whose name is prefixed by '–' will not cause the commands for the high-level event to be executed, although the token will still be queued on arrival. Hence, only the arrival of a token from `'pop` triggers execution of `'push_pop`; the arrival of a token from `'push` does not.

The code associated with `'push_pop` reflects the last-in, first-out stack discipline. It maintains the following invariant on its token queues: Whenever it is not active, its `'pop` queue is empty and its `'push` queue contains, in arrival order, all `'push`'s that have not yet been matched with a `'pop`. The code first checks that the `'stack_p` attributes passed on from the most recent `'push` and `'pop` are the same. These attributes reflect the values of *stack_pointer* when the item was originally pushed and later popped. Should their values differ, `'push_pop` prints an error message. Otherwise, it prints the recorded *stack_pointer* value and the contents of the top of stack. The code uses the pre-defined function `$value()` to access its constituents' tokens' attributes. `$value()`'s three arguments specify the name of the constituent, the position in the high-level event's queue for that constituent, and the attribute name. (Queue positions are numbered starting with 1 denoting the oldest arrival; as a convenient special case, 0 denotes the most recent arrival.) After checking the `'stack_p` attributes, `'push_pop` removes the corresponding constituent events' tokens from its queues using the **event remove** command, which uses the same position numbering as `$value()`.

```
# Raise 'push when push entered
break push
commands
    silent
    event raise 'push
    cont
end

# Raise 'pop when pop entered
break pop
commands
    silent
    event raise 'pop
    cont
end
```

Fig. 5. Activating the primitive stack events.

As seen in Fig. 5, the primitive events `‘push` and `‘pop` are activated by an **`event raise`** command from within breakpoints set inside the *`push`* and *`pop`* functions respectively. After the **`event raise`** command is executed and the dataflow graph once again becomes quiescent, the **`cont`** command instructs the inferior process to continue executing. Fig. 6 shows the dataflow graph corresponding to the code presented in Figs. 4 and 5.

Memory Allocation

This example uses events to monitor allocations and deallocations of memory. Misuse of memory allocation and deallocation routines (*`malloc`* and *`free`* in C terminology) is a common source of program error, e.g., passing the deallocation routine an invalid address, especially one for a block that was previously freed. This example shows how errors in the use of the memory allocator can be detected.

Fig. 7 shows the definition of two primitive events, `‘malloc` and `‘free`, which correspond to invocations of the allocation and deallocation routines. `‘malloc`

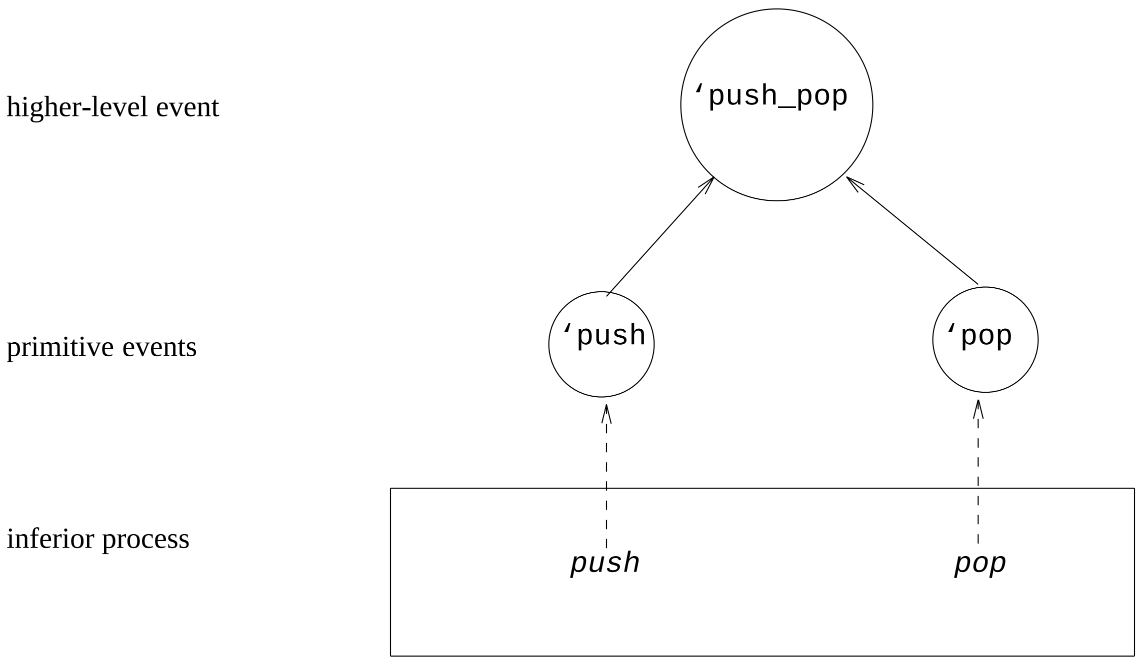

Fig. 6. Dataflow graph for stack example.

```
# The 'malloc event is recognized each time it is triggered/raised
# by default; the same applies to the 'free event.
event define 'malloc ('address)
    event set-attribute 'address $quiet_finish()
end

event define 'free ('address)
    event set-attribute 'address $addr
end

# The high-level event 'match depends on lower level events 'malloc and 'free
# but it is triggered only by 'free.
event define 'match ()   -'malloc  'free
   {
   local $qindex, $free_addr
   # $qlen() returns the length of specified event queue
   set $qindex = $qlen('malloc)
   set $free_addr = $value('free, 1, 'address)
   while ($qindex > 0)  # Search for 'malloc matching this 'free
       if ($value('malloc, $qindex, 'address) == $free_addr)
           # Remove first token from 'free's queue
           # and $qindex'th token from 'malloc's
           event remove 'free, 1
           event remove 'malloc, $qindex
           wbreak   # Break out of the while loop
       endif
       set $qindex--
   endwhile
   if ($qindex == 0)
       printf "Error: attempt to free unallocated address 0x%x\n",\
              $free_addr
       event remove 'free, 1
       event dont-propagate
   endif
   }
end
```

Fig. 7. Memory allocator monitor.

records in its attribute `'address` the address of the memory block allocated; `'free` records in its attribute `'address` the address of the memory block to be deallocated.*

Error detection is carried out in the high-level event `'match`. `'match` keeps track of the addresses of all memory blocks that have been allocated but not yet matched with a deallocate request. Whenever `'free` occurs, `'match` checks to see if the corresponding `'malloc` event has been raised. If a `'malloc` with such an address cannot be found, `'match` prints an error message. Note that the order of freeing blocks need not be the same as that in which they were allocated.

As shown, `'match` is triggered by `'free`. The matching of `'malloc` and `'free` events is performed using a simple **`while`** loop, which sequentially compares the attribute `'address` passed by `'free` with those returned by the previous `'malloc` events. This example also illustrates a local block, enclosed within '`{`' and '`}`', that declares two local convenience variables, `$qindex` and `$free_addr`.

This example can be extended to determine whether the reason for a mismatched *`free`* is that the block is being freed a second time. To do so, `'match` requires only small modifications. An attribute, say `'maddr`, needs to be specified in its definition and set when a match is found. Also, a loop needs to be added to the body of the second **`if`** statement. The loop simply searches the event history list for a `'match` whose `'maddr` attribute is equal to `$free_addr`. With additional attributes in `'malloc` and `'free` recording the names of the inferior process's procedures that invoke *`malloc()`* and *`free()`*, the crucial transition from merely detecting the symptoms of a bug to discovering its cause can be made. Searching the history list is also useful in different contexts. Suppose, for example, that the user has detected that the contents of an allocated structure are corrupted. Then, a programmatic search of the history list can determine whether the address of that structure was freed.

Tree Traversal

The example shown in Fig. 8 illustrates a simple user-defined function, `$ptree()`, that outputs in-order the nodes of a binary tree. The user may invoke `$ptree()` interactively from Dalek's command level or from any point during execution of the inferior process by simply setting a breakpoint whose commands call `$ptree()`. A function such as `$ptree()` is especially useful to invoke in the

---

* `$quiet_finish()` is a Dalek pre-defined function that resumes execution of the inferior process until it returns from the current procedure. `$quiet_finish()` returns the procedure's return value. `$addr` is a global convenience variable set inside *`free`* to the address of the block being freed (i.e., *`free`*'s argument). Since *`free`* is a system-dependent library routine, how `$addr` is set is also system-dependent.

```
#  Dalek code to print out a binary tree;
#  its argument is a pointer to a subtree.
function $ptree ($p)
    if ($p != 0)
        call $ptree($p→left)
        printf "%c\n", $p→item
        call $ptree($p→right)
    endif
end
```

Fig. 8. Tree traversal in Dalek.

routines that manipulate trees—e.g., those that insert or delete nodes—since it displays their effects on the tree. Note that functions such as `$ptree()` are written in Dalek and are executed in the debugger's address space. No modification of the user program is required.

Simulation Program

This example shows the usefulness of Dalek in debugging a pseudo-concurrent simulation program. Pseudo-concurrent programs, though deterministic, are similar to true concurrent programs. They consist of multiple simulated processes executing in parallel and sharing global variables. Furthermore, the exact interleaving of process execution is hard to predict. Such programs are obviously difficult to debug.

The simple simulation program shown in Fig. 9 uses the CSIM simulation package [CSIM87]. It consists of six processes: one is the "master", each of the others is a "sender". Each sender repeatedly sends a message back to the master; the delivery of each message takes a random period of time.

One difficulty in debugging such a program is the large number of interactions between processes. It is difficult to isolate the relevant interactions. A common practice is to insert into the source program debugging statements to print out information about the current state of the process. However, these debugging messages will be intermixed in pseudo-concurrent programs. Filtering out irrelevant messages can be tedious. Using Dalek, however, such filtering is easily accomplished. Fig. 10 shows two primitive events, `‘send` and `‘recv`, and the higher-level event `‘message`. `‘send` is raised when a message is sent by sender 3; `‘recv` is raised when the master receives a message from sender 3. (For simplicity, the sender number is encoded in the message.)

```
#include "lib/csim.h"           /* defines CSIM types, procs, vars: MBOX,
                                      EVENT, event, create, clock, etc. */
#define STM 2.0                 /* mean send time */
#define NUM 50                  /* number of items */
#define SENDERS 5               /* number of senders */
MBOX mb;                        /* mail box for messages */
EVENT go_ahead;                 /* tell sender to start */
sim() { int i, msg;
    mb = mailbox("mail box");
    create ("master");          /* make this a CSIM process */
    go_ahead = event ("Start sending message");
    for (i=1; i<=SENDERS; i++)
        sender();               /* initiate processes */
    hold (1.0);                 /* wait for creation of all senders */
    set (go_ahead);             /* signal all senders to begin */
    for (i=1; i<=NUM; i++) {
        receive (mb, &msg);     /** NOTE: Dalek event 'recv is raised here **/
        printf ("Received %d from %d\n", msg % 100, msg / 100);
    }
}
sender () { int i, me;
    create ("sender");          /* make this a CSIM process */
    me = identity();
    wait (go_ahead);
    for (i=1; i<=NUM/SENDERS; i++) {
        printf ("Sent %d from %d\n", i, me);
        hold (expntl(STM));     /* hold message transmission time */
        send (mb, me*100+i);    /** NOTE: Dalek event 'send is raised here **/
    }
}
```

Fig. 9. A simple simulation program in C.

```
# Events 'send and 'recv are raised from breakpoints as indicated in Fig. 9;
# each propagates (recognizes) only messages from sender number 3.
event define 'send ('from, 'msg)
    if (me == 3)
        event set-attribute 'from me
        event set-attribute 'msg i
    else
        event dont-propagate
    endif
end
event define 'recv ('from, 'msg)
    if (msg / 100 == 3)
        event set-attribute 'from msg/100
        event set-attribute 'msg msg%100
    else
        event dont-propagate
    endif
end

# Correlate 'send and 'recv by checking message contents
event define 'message () –'send 'recv
    if (  ($value('send, 1, 'from) == $value('recv, 1, 'from)) && \
          ($value('send, 1, 'msg) == $value('recv, 1, 'msg))  )
        printf "Message %d from %d completed. Time=%lg\n",\
            $value('send,1,'msg),\
            $value('send,1,'from), clock
    else
        printf "Error:: sender: from=%d, msg=%d,", \
            $value('send, 1, 'from), $value('send, 1, 'msg)
        printf "receiver: from=%d, msg=%d\n", \
            $value('recv, 1, 'from), $value('recv, 1, 'msg)
    endif
    event remove 'send, 1   # Remove first token from queue for 'send
    event remove 'recv, 1   # and first token from queue for 'recv
end
```

Fig. 10. Dalek code to filter and match sends and receives.

`'send` and `'recv` discard all other messages. When `'message` matches a pair of `'send` and `'recv`, it outputs information regarding the match. If `'message` detects a difference in the contents of the message sent from that received, it prints an error message. As indicated in Fig. 9, `'send` and `'recv` are raised from breakpoints associated with the source code. The commands for these breakpoints are similar to those in Fig. 5 and are therefore not shown.

DISCUSSION

Comparison with Other Debugging Work

A general event-based approach to debugging has not been used in any viable sequential debugger. A number of debugging tools (e.g., see References [Elshoff88], [Lin88], [Bates82], [Bates83], and [Bates88]) for distributed programs have employed this approach. However, most of these tools only look at the interactions between processes and do not provide information about the internal state (such as content of variables, stack frames, etc.) of individual processes. Thus, they cannot be used for debugging sequential programs or sequential parts of distributed programs. On the other hand, common sequential debuggers—*DEBUG*, *dbx*, *gdb*, *sdb*—require programmers to think solely in terms of breakpoints, which are too low-level and cannot easily be correlated with the abstractions present in the source program. In Dalek, events can group together data from breakpoints or other events into useful abstractions. The use of a programming language as the debugging command language also gives more complete control over the execution of the inferior process and the analysis of its behavior.

One contribution made by Dalek is that its event recognition is an active process, which, depending on the conditions and actions specified by the user, can modify the state of the inferior process in any desired way. Many existing event-based debuggers can only passively record all or some of the events generated by the inferior process, and then replay them in some controlled manner. Information that is not recorded directly, or cannot be derived from the recorded events, is unavailable. Dalek event specification allows association of arbitrary debugging commands with each event. The recognition of an event may cause Dalek to interact with the inferior process, either to obtain more information or to modify the inferior process's execution. Thus Dalek provides considerably more power to control and examine the inferior process's behavior.

Many event-based debuggers employ pattern matching approaches to event recognition. These approaches often require special expressions (e.g., see References [Bates88] and [Bruegge83]) to describe the patterns to be matched. These expressions are sometimes complicated and confusing, and can only recognize those sequences of events that exactly match the given patterns. As a consequence, erroneous event sequences are hard to detect because their patterns are often not known in advance. Dalek's dataflow approach allows greater flexibility in event recognition. Each non-primitive event (i.e., interior node in the dataflow graph) is a computational unit and has access to all the attributes associated with the lower-level events on which it depends. Based on this information, an arbitrarily complicated condition can be evaluated before deciding whether the event should be recognized and propagated to higher levels. As a result, both normal and erroneous event sequences can be captured.

The dataflow approach to event-based debugging has several significant advantages over traditional debuggers. First, it allows correlation of breakpoints and events, as illustrated in Fig. 1. Moreover, the correlation activity can depend on information in the event queues, as shown in the stack example. While such correlation can be accomplished using some more traditional procedural debuggers, the techniques are at best ad hoc and cumbersome. Second, the dataflow approach allows debugging code to be written in blocks that are generally independent. For example, consider again the breakpoints and events in Fig. 1. Suppose that a new high-level event, K, is to depend on primitive events D and F and a new primitive event L, which is raised at a new breakpoint M. This change requires the user to write breakpoint code for M, primitive event code for L, and high-level event code for K. The code for D and F need not be changed unless K needs new attributes from them. Under traditional procedural debuggers, making a change equivalent to adding high-level event K would require substantial rewriting of the debugging code; in particular, the code associated with each breakpoint would need to be modified.

A third advantage of the dataflow approach is that it provides a history list and mechanisms to access it programmatically. Thus, program behavior can be explored after the program has executed. The extension to the memory allocator example described one typical use of the history list; see Reference [Crawford90] for other uses. Using more traditional procedural debuggers, new debugging procedures would need to be defined and bound to breakpoints (being careful not to interfere with existing breakpoints at the same locations), and then the inferior program would need to be re-executed.

Dalek provides a framework for obtaining a high-level abstraction of the behavior of the program, but it does not create such abstractions by itself. Dalek is not an expert system that performs the debugging *for* the users. Rather, the user must specify what the primitive events are, where they are to be raised, and how they should be grouped together to form high-level events. This property is shared with all conventional debuggers, although some current work [Bates8811] might offer more help.

Although many existing sequential debuggers allow examination and modification of the state of the inferior process, the expressiveness of their commands is extremely limited and their syntax is somewhat ad hoc. In many situations, a more powerful and flexible command language is necessary to describe the desired action. Dalek's approach of using a complete programming language has proved to be very useful to this end. For example, under most existing sequential debuggers, printing or modifying values in nodes of a linked list of N elements would require the user to type N separate commands (although with macros and last-command repetition, this task would merely be tedious rather than onerous). The same operation can be done in Dalek (or DEBUG) with a simple loop. Dalek's approach here is similar to that in References [Johnson78] and [Johnson81]. The key difference is that Dalek includes these mechanisms as part of a conventional (and modern) debugger rather than in an interpretive environment.

Many programmers augment their programs with extra debugging statements and run-time assertion checks, which are often not needed in production versions. As was illustrated with the `$ptree()` function, Dalek provides a simple means to write such code as the need arises and to control when it is executed. Dalek can be used similarly to specify, during the debugging of a program, new assertions to be checked. An advantage of defining such statements and assertions in Dalek's address space is that their debugging variables are not susceptible to corruption by the inferior process.

The Dalek language is powerful enough to replace entire functions or statements in the inferior process. A user can fix a bug (such as forgetting to check an array bound before accessing an array element) by, in effect, replacing the bad code with instructions written in the debugger language. (A **`jump`** command will skip around the bad code and then resume execution of the inferior process.) In this way, multiple errors can be found and fixed without exiting the debugger and recompiling the program.

The memory allocation example was actually used to find an insidious memory allocation bug in Dalek itself. That is, Dalek was used to debug itself. Using Dalek to monitor all the memory allocation and deallocation for a large complicated program like Dalek took a fair amount of time. Dalek required about one minute of elapsed time on a moderately-loaded (load average about 6) VAX 8600 before the inferior Dalek finished initializing its data structures; the overhead for the rest of the monitoring, while noticeable, was acceptable. In that one minute, Dalek handled 399 events generated by 369 *`mallocs`* and 15 *`frees`*; i.e., 369 `'malloc`, 15 `'free`, and 15 `'match` events were triggered. Using Dalek for such debugging is certainly more convenient and much faster (even given that the implementation of Dalek's event handling is far from optimal) than manually checking the arguments in each invocation of *`malloc`* and *`free`*.

Dalek has also been used as a performance measuring tool. Programmers usually obtain timing information for a particular segment of code by explicitly inserting system calls in their programs to record the time before and after that segment executes, and then printing the difference. Dalek provides a pre-defined function that returns the cpu time used by the inferior process. Hence, the execution time of the inferior process can be measured without altering its source code. Such timing is specified as part of an event or breakpoint, and therefore can easily be changed interactively and dynamically. This timing measurement is accurate for user time but not for system time.* (User time is the time the cpu spends executing the instructions in the user process's address space, whereas system time is the time the cpu spends executing system calls invoked by the process.) The reason for this discrepancy is that some of the cost of the debugger controlling the inferior process is charged as system time to the inferior process. Dalek can also be used as a simple code profiler and to gather other useful statistics concerning program behavior. For example, Dalek has been used to count the number of times a given procedure or statement is executed, to find the maximum length a linked list reaches during execution, and to measure run-time stack frame behavior in order to determine conditional probabilities that a push follows another push, a pop follows a push, etc. As above, Dalek can obtain such information without the user needing to modify or recompile the inferior process's source code.

One possible objection to the Dalek approach is that it requires the programmer to learn 'yet another language'. Although the Dalek language differs from C, the language in which inferior programs are written, experience indicates that C programmers have little trouble learning it. The main reason that the Dalek language differs syntactically from C is that the primary concern during Dalek's development was on *semantic* issues and functionality rather than on 'syntactic sugar'. The way in which the implementation evolved also favored a non-C flavor. In hindsight, the Dalek language *could* have been made syntactically closer to C. Whether that is a good idea and worth further development effort is unclear at this time.

Debugging the Debugging Code

The question naturally arises: With any debugger language as powerful as Dalek's, what should one do if some debugging code written in that language is itself suspected of containing errors? How can instructions written in the debugger language be debugged? It might seem that undertaking two-stage debugging is a possible solution, but in reality such tandem-debugging only complicates matters even further. Early in Dalek's development, one copy of Dalek was used to help debug Dalek language code being interpreted by another copy of Dalek, which was in turn overseeing the execution of an application program. That experience indicated unequivocally that such two-stage debugging is not a viable solution. Instead, Dalek supports several "introspection" facilities so the user can interrogate it regarding its state.

* User time is not always accurate on all machines. For example, on machines that employ instruction pipelining, it can be affected by what instructions are in the pipeline; the pipeline is flushed when the operating system switches contexts, which occurs in alternating execution between the inferior process and the debugger.

One Dalek introspection facility is verbose mode, which can be selectively enabled by the user. In this mode, Dalek displays every command (and the name of its enclosing block) as it is interpreted. Making the debugger's actions visible in this way allows many simple errors to be found easily.

Dalek also provides several means to pause its execution during interpretation of the user's debugging program. The user can then examine or alter the state of the inferior process and, equally importantly, that of the debugging program itself. When the debugging program is paused inside an event, for example, values of that event's attributes can be manually inspected and altered. The debugging program's logic can be overridden by breaking out of an enclosing **`while`** loop or by forcing a user-defined debugging function to prematurely return a specified value. In addition, any block of code in the debugging program can be edited before resuming interpretation.

Dalek will pause in its interpretation of debugger commands when it encounters a **`pause`** command or when the user types the interrupt character, e.g., control-C. Using this latter method obviously does not afford the user much choice over the line of debugging code at which interpretation pauses. To pause at a particular line of debugging code, the user can set a meta breakpoint at any line in the Dalek language program.

For tighter control, "dstep" mode allows meta single-stepping. Prior to interpreting each line of debugging code, dstep mode displays the same information as is shown in verbose mode, but in addition it offers the user the option of pausing, editing the current command block, or skipping the displayed command entirely.

Status and Future Work

All of the Dalek features described in this paper and several other features have been fully implemented.* Dalek's implementation augments *gdb*'s (version 3.0) approximate 35,800 lines of C source code by roughly 9,400 lines. (See References [Crawford90] and [Olsson90] for more details on Dalek's implementation.) The Free Software Foundation plans to incorporate Dalek into a future release of *gdb.*

As a prototype for demonstrating the feasibility of new ideas, Dalek does not include all features that an ideal debugger should have, and its implementation is not optimized. However, experience with Dalek has indicated that even the prototype is a significant improvement over existing sequential debuggers. It has also served to highlight a number of areas that deserve further study.

---

* The single exception is that convenience variables are restricted to have types that have been declared in the inferior process; new types cannot presently be defined within Dalek.

One needed improvement is to raise the level of the Dalek language. The current language is intentionally low-level to ensure sufficient power and flexibility for experimentation. However, its low level sometimes makes it cumbersome to use, e.g., the way that the event queue is searched in the memory allocation example (Fig. 7) requires having to code a loop to examine each value in the event queue. Experience with Dalek is identifying which primitive mechanisms are most commonly used and how they are commonly combined. These common combinations will be made available to the user as higher-level mechanisms in the debugger language.

Dalek also provides an extremely limited form of replay mechanism that is actually a shorthand for re-executing the program from the beginning to the specified point. Although replay and graphical displays are useful debugging features [Garcia85], [LeBlanc85], [LeBlanc87], [Lin88], and [Adams86], the authors' future efforts will focus on the other, more fundamental issues.

Dalek currently works only on programs written in C. It could be extended to work on programs written in other languages. Techniques that provide other debuggers with such multilingual capabilities—such as those used in *DEBUG* [Beander83] and *SWAT* [Cardell83], and those planned for future versions of *gdb*—could be used. As mentioned above, however, immediate future efforts will focus on other areas.

One area of current work is to extend Dalek to support the debugging of concurrent programs. In fact, the dataflow approach to event-based debugging was initially developed with the intention of working within distributed environments. Event-based debugging has the advantage of pulling together related information that is scattered among different contexts or is only available at separate points in time, which is exactly the case in distributed programs. For example, a message send and a message receipt take place at different times and in different process's contexts, but they group naturally into one higher-level event (as illustrated in the simulation example).

CONCLUSION

This paper has presented a new approach to event-based debugging. The two key points in the approach are modeling events using a dataflow graph and providing a debugging language that is fully programmable. This paper has also described how that approach has been incorporated into a sequential debugger, Dalek. Dalek's implementation has demonstrated the feasibility of the approach and has provided a base for hands-on experience. That experience has indicated that Dalek is a significant improvement over existing sequential debuggers. The methodology described in this work provides a solid foundation on which to base debugging activities. The tools described in this work represent concrete progress toward addressing some of the significant problems of debugging sequential programs. The methodology can be extended to the realm of debugging concurrent programs; incorporating the tools into debuggers for concurrent programs could substantially improve their functionality.

ACKNOWLEDGEMENTS

Richard Stallman and the Free Software Foundation (FSF) make *gdb* available to the public. Having such a good base on which to build Dalek greatly expedited this research. Chris Wee, Mudita Jain, Carole McNamee, Cui Zhang, and the anonymous referees provided very useful comments on Dalek and on earlier drafts of this paper. Early work on this research was supported in part by System Integrators, Incorporated (SII) and the State of California under the MICRO program.

REFERENCES

1. F. Baiardi, N. De Francesco, E. Matteoli, S. Stefanini, and G. Vaglini, 'Development of a debugger for a concurrent language', *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, CA, March 1983, pp. 98-106.
2. R. Curtis and L. Wittie, 'BugNet: A debugging system for parallel programming environments', *Proc. Third International Conference on Distributed Computing Systems*, Hollywood, FL, October 1982, pp. 394-399.
3. I.J.P. Elshoff, 'A distributed debugger for Amoeba', *Proc. Workshop on Parallel and Distributed Debugging*, Madison, WI, May 1988, pp. 1-10.
4. J. Gait, 'A debugger for concurrent programs', *Software—Practice and Experience*, **15**, (6), 539-554 (1985).
5. M.E. Garcia and W.J. Berman, 'An approach to concurrent systems debugging', *Proc. Fifth International Conference on Distributed Computing Systems*, Denver, CO, May 1985, pp. 507-514.
6. H. Garcia-Molina, F. Germano, and W.H. Kohler, 'Debugging a distributed computing system', *IEEE Trans. Softw. Eng. SE-10*, (2), pp. 210-219 (1984).
7. P.K. Harter, D.M. Heimbigner, and R. King, 'IDD: An interactive distributed debugger', *Proc. Fifth International Conference on Distributed Computing Systems*, Denver, CO, May 1985, pp. 498-506.
8. R.J. LeBlanc and A.D. Robbins, 'Event-driven monitoring of distributed programs', *Proc. Fifth International Conference on Distributed Computing Systems*, Denver, CO, May 1985, pp. 515-522.
9. T.J. LeBlanc and J.M. Mellor-Crummey, 'Debugging parallel programs with instant replay', *IEEE Trans. Comp. C-36*, (4), 471-482 (1987).
10. C. Lin and R.J. LeBlanc, 'Event-based debugging of object/action programs', *Proc. Workshop on Parallel and Distributed Debugging*, Madison, WI, May 1988, pp. 23-34.
11. R.L. Schwartz, P.M. Melliar-Smith, and F.H. Vogt, 'An interval logic for higher-level temporal reasoning', *Proc. Second Annual Symposium on Principles of Distributed Computing*, Montreal, Quebec, August 1983, pp. 173-186.
12. E.T. Smith, 'A debugger for message-based processes', *Software—Practice and Experience* **15** (11), pp. 539-554 (1985).
13. L. Wittie and R. Curtis, 'Time management for debugging distributed systems', *Proc. Fifth International Conference on Distributed Computing Systems*, Denver, CO, May 1985, pp. 549-550.
14. E. Adams and S.S. Muchnick, 'Dbxtool: A window-based symbolic debugger for Sun workstations', *Software—Practice and Experience*, **16**, (7), 653-669 (1986).
15. J.F. Maranzano and S.R. Bourne, 'A tutorial introduction to ADB', in *UNIX Programmer's Manual, 7th Edition, Vol. 2A* Bell Laboratories, Holmdel, NJ, January 1979.
16. *VMS Debugger Manual*, AA-LA59A-TE, VMS version 5.0, Digital Equipment Company, Maynard, MA, April 1988.

17. ‘DBX (1)’, in *UNIX Programmer’s Manual*, 4.2 Berkeley System Distribution, Vol. 1, Computer Science Division, University of California, Berkeley, CA, August 1983.

18. H. P. Katseff, ‘Sdb: a symbolic debugger’, in *UNIX Programmer’s Manual, 7th Edition, Vol. 2C* Bell Laboratories, Holmdel, NJ, January 1979.

19. R.M. Stallman, *GDB Manual (The GNU Source-Level Debugger), Third Edition, GDB version 3.1*, Free Software Foundation, Cambridge, MA, January 1989.

20. P. Bates and J.C. Wileden, ‘EDL: A basis for distributed system debugging tools’, *Proc. Fifteenth Hawaii International Conference on System Sciences*, Honolulu, HI, 1982, pp. 86-93.

21. P. Bates and J.C. Wileden, ‘An approach to high-level debugging of distributed systems (preliminary draft)’, *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, CA, March 1983, pp. 107-111.

22. P. Bates, ‘Debugging heterogeneous distributed systems using event-based models of behavior’, *Proc. Workshop on Parallel and Distributed Debugging*, Madison, WI, May 1988, pp. 11-22.

23. J. Tulloch and M. Alvarado, *Doctor Who—The Unfolding Text*, Macmillan Press, 1983.

24. R.H. Crawford, ‘Topics in Behavioral Modelling and Event-Based Debugging’, M.S. Thesis, Div. of Computer Science, University of California at Davis, in preparation.

25. R.A. Olsson, R.H. Crawford, and W.W. Ho, ‘Dalek: a GNU, improved programmable debugger’, *Proc. Summer 1990 USENIX Conference*, Anaheim, CA, June 1990, pp. 221-231.

26. *CSIM Reference Manual (Revision 11)*, Microelectronics and Computer Technology Corporation, Austin, TX, ACA-ST-252-87, October 1987.

27. B. Bruegge and P. Hibbard, ‘Generalized path expressions: A high level debugging mechanism’, *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, CA, March 1983, pp. 34-44.

28. P. Bates, private communication, November 1988.

29. M. S. Johnson, ‘The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment’, Ph.D. Dissertation, The University of British Columbia, TR-78-6, August 1978.

30. M. S. Johnson, ‘DISPEL: a run-time debugging language’, *Computer Languages*, **6**, 79-94 (1981).

31. B. Beander, ‘VAX DEBUG: An interactive, symbolic, multilingual debugger’, *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, CA, March 1983, pp. 173-179.

32. J.R. Cardell, ‘Multilingual debugging with the SWAT high-level debugger’, *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, CA, March 1983, pp. 180-189.