Norman Ramsey

Dalek: A GNU, Improved Programmable Debugger

Ronald A. Olsson
University of California at Davis
Division of Computer Science
Davis, CA 95616
olsson@ivy.ucdavis.edu

Richard H. Crawford
University of California at Davis
Division of Computer Science
Davis, CA 95616
crawford@ivy.ucdavis.edu

W. Wilson Ho
University of California at Davis
Division of Computer Science
Davis, CA 95616
how@ivy.ucdavis.edu

ABSTRACT

Dalek is a debugger for UNIX systems that offers considerably more power than existing UNIX debuggers. Dalek achieves this by providing a powerful debugging language and events. The Dalek debugging language provides mechanisms comparable to those found in conventional programming languages, i.e., conditional and looping statements, blocks, local variables, procedures, and functions. The user associates arbitrary Dalek code with breakpoints in a program—e.g., to print out the contents of a binary tree or to ensure that a linked list is circular. Dalek events provide a way to form higher-level abstractions during the execution of a program. Low-level events (including primitive events such as those that can be raised at breakpoints) can be combined into higher-level events that correspond more closely to the abstractions present in a program; the programmer need not be concerned with the occurrences of the low-level events, only with their net effect as a high-level event. As an example, the low-level events of a push and a pop on a stack can be combined into one higher-level event. Dalek's method of combining events is novel: it is similar to coarse-grained dataflow, where each token corresponds to an instance of an event and nodes are used to combine low-level events into higher-level ones. Tokens can carry user-specified attributes to distinguish among different occurrences of the same event. Nodes in the dataflow graph contain Dalek code by which the user states how to combine events. Dalek's approach is more powerful than existing schemes, which typically employ pattern matching. The mechanisms Dalek provides also facilitate its use as a performance measurement tool and for patching code on the fly.

A prototype implementation of Dalek has been operational since summer 1988. It is based on *gdb*, the Free Software Foundation's GNU project debugger, and enhances *gdb*'s functionality with the additional language and event mechanisms described above. Dalek is being incorporated into a future release of *gdb*. This paper describes Dalek, its implementation, and experience using it.

1. Introduction

Existing debuggers for sequential compiled UNIX programs (e.g., *adb* [Maranzano79], *dbx* [DBX83], *sdb* [Katseff79], and *gdb* [Stallman89]) require the user to interact at a very detailed level instead of at the level of the abstractions present in the user's program.[1] Such debuggers typically provide users with a breakpoint facility, which allows them to suspend execution and examine variables when control reaches a specified point in the inferior process.

Current UNIX debuggers have two significant inadequacies.

- They provide only very limited control over the actions taken when a breakpoint occurs. At best, they provide only conditional execution of the entire block of commands associated with a breakpoint. More powerful and discriminating means of control are desirable, e.g., a loop to print out the contents of a linked list, or a loop to single-step the inferior process until one variable is greater than another.
- They provide the user 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.

Our debugger, Dalek[2], remedies both of these inadequacies. Dalek's language provides conditional and looping statements, blocks, local variables, procedures, and functions. This approach is similar to that in [Johnson78] except Dalek includes these mechanisms as part of a conventional (and modern) debugger rather than in an interpretive environment. Dalek also provides support for *events*—occurrences of interesting activities in the execution of the inferior program—as a way to form higher-level abstractions during the execution of a program. Dalek employs a novel, coarse-grained dataflow approach for combining events in which the dataflow graph's nodes contain fragments of code written in the Dalek language. Event-based debugging has received much attention for debugging concurrent programs (e.g., [Baiardi83], [Bates82], [Bates83], [Bates88], [Elshoff88], and [Lin88]), but it has not been applied to debugging sequential programs. The mechanisms Dalek provides also facilitate its use as a performance measurement tool and for patching code on the fly.

The rest of this paper is organized as follows. Section 2 gives an overview of Dalek, using several examples to illustrate Dalek's usefulness and power, and discusses design and implementation issues that were fundamental in Dalek's development. Section 3 evaluates Dalek from the perspectives of implementation effort, performance, and lessons learned. Finally, section 4 contains some concluding remarks, including the direction our work on debugging is taking.

2. The Dalek Solution

In this section, we introduce Dalek and demonstrate by means of examples how it can be used to solve common debugging problems. We also discuss related design and implementation issues. (See [Crawford89,Crawford90] for additional details, further examples, and other goodies.)

---

[1] 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 program, which executes as a separate, 'inferior' process.

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

2.1. Overview of Dalek

Dalek is based on *gdb* (version 3.0). 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[3], procedures, and functions. It also supports events, an abstraction mechanism used to construct models of the inferior program's behavior (either its intended or undesirable forms) and to monitor the program's actual behavior in terms of the specified forms.

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** (short for 'continue') command resumes execution of the inferior process. The **step** command executes the next statement of the inferior process. Names of convenience variables and Dalek's built-in and user-defined functions start with a '$'. Names of events and their attributes start with a backquote, e.g., 'foo. Comments begin with a '#'.

2.1.1. Dalek Programming

Dalek code can be specified dynamically as part of the debugging activity, without requiring the source code to be modified, recompiled, and relinked. Besides avoiding the overhead of those activities, this approach has the additional advantage that the user can specify what is of interest as s/he gleans information during the inferior's execution, i.e., for "program exploration".

Dalek code can be executed from the debugger's command level or automatically at a breakpoint or when an event is triggered. It can be entered interactively or read from a file.

The Dalek language contains all the mechanisms that programmers generally find useful: assignment statements, **if** and **while** statements, blocks, local convenience variables, procedures, and functions. 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. To illustrate the Dalek language, consider the Dalek code shown in Figure 1. The code defines a $print_list() procedure. The commands **push-block**, **pop-block**, and **create-local** are used together to effect a local scope. This procedure can then be invoked interactively from Dalek's command level or from any point during execution of the inferior process by simply setting a breakpoint whose commands include a call to the procedure. As a simple example, $print_list() can be invoked from a breakpoint at line 27 of file *goo.c* by the commands

---

[3] A convenience variable is a variable that the user defines and manipulates within the debugger; *gdb* provides global convenience variables.

```
# Dalek procedure to print out value field of nodes in
# a circularly linked list starting at $head.
# Assumes $head is a dummy node,
# whose next field points to itself if the list is empty.
function $print_list($head)
    push-block
    create-local $p
    set $p = $head->next
    while ($p != $head)
        printf "%d\n", $p->value
        set $p = $p->next
    endwhile
    pop-block
end
```

Figure 1. A Dalek procedure to print a linked list.

```
break goo.c : 27
commands
    silent # suppress the default announcement of encountering a breakpoint
    call $print_list(symtab_head) # print list headed by symtab_head
    cont # continue execution of the inferior process
end
```

Note that the body of $print_list() could have been written inline as commands in the above. However, it is written as a function so it can be invoked directly from the command level or from within another breakpoint, if desired.

In a manner similar to the above $print_list() example, Dalek code can be written to ensure that the inferior maintains certain invariants regarding its data structures. For example, code to check that a circularly linked list is indeed circular could be invoked on entry to and exit from each procedure that manipulates the list. This approach is more flexible than using compiled-in assertion checks (e.g., like the *assert* macro in C) since what invariant to check and where to check it can be decided when actually debugging the inferior.

To demonstrate further how Dalek's mechanisms can be used to attack a problem, we describe our experience using Dalek to find a bug in code generated by lex(1). The execution of the lex-generated code resulted in a segmentation fault occurring in a reference to a table element within the procedure *yylex*. Setting a breakpoint at procedure *yylex* and reexecuting the inferior showed that the procedure was a frequently executed one. We then used a counter to determine the particular invocation of *yylex* that resulted in the segmentation fault. Next we modified the breakpoint code so it would simply continue execution of the inferior for all invocations of *yylex* except the last one. For the last invocation, it would single step the inferior as long as the reference that was causing the segmentation fault was within the bounds of the table. At this point, the Dalek code looked something like that shown in Figure 2. Executing the inferior in conjunction with this debugging code thus showed the first point at which the condition went out of range. From examining the inferior's code, it was then easy to see that it was using a byte (char) for indexing a large table, and the byte had overflowed.

```
set $count = 0
break lex.yy.c : yylex
commands
   silent # suppress the default announcement of encountering a breakpoint
   if( ++$count < $N )  # $N was determined earlier
      cont # continue execution of the inferior process
   else # single step the inferior while valid reference.
      #       note:  yyt->advance is used in the source statement
      #       yystate = yyt->advance + yysvec
      while( yyt->advance >= 0 )
         step
      endwhile
      printf "%d before start of table\n", yyt->advance
   endif
end
```

Figure 2. Dalek code for isolating the cause of a segmentation fault.

The iterative technique used above to isolate a bug is not new. What is interesting, however, is that Dalek gives the right tools for assisting the user in this process. This problem would be more difficult to attack using other debuggers since they do not provide **while** statements and/or convenience variables. Using *dbx*, for example, variables can be traced so that information is output when their values change; further, that output can be conditional. However, the amount of output that could be printed before the relevant output appeared would most likely overwhelm the user.

2.1.2. Dalek Events

An event in Dalek is the occurrence of an interesting activity in the execution of the inferior program. Each event can have any number of dynamically-typed attributes associated with it. 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. As with other Dalek language commands, event definitions can be given interactively or read from a file.

2.1.2.1. Dataflow-based Event Recognition

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 triggers a primitive event, which, if it successfully completes its recognition code, generates a token. Since a token corresponds to an *instance* of an event, a token is a composite 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 the token characterizing the event 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. The definition of

a high-level event specifies which of its constituents will trigger that high-level event. That is, the definition indicates which tokens, when 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 in the definition of the high-level event indicates that the arrival of that constituent should *not* trigger the high-level event.)

When a high-level event is triggered, its associated code decides how it should react based on what constituents are present and their values. 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, 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 constituents from its incoming queues so as to avoid repeated recognition of the same tokens. When the high-level event's code finishes execution, tokens embodying the new event instance 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.

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 undefined. When all triggered nodes have executed—i.e., the graph reaches a quiescent state—Dalek continues execution with the command following the one that raised the initial primitive event.

The exception to the above general rule arises when a node executes a command that resumes the inferior process. 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 resuming execution of the inferior process might cause it to encounter another breakpoint, which can cause another primitive event to be raised. Dalek handles such events by stacking their executions; the details are described in [Crawford90].

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.

2.1.2.2. Primitive Events

In order for an instance of a primitive event to occur, it must be explicitly raised. Typically, the user specifies an **event raise** command in the command block of a breakpoint inserted 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 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.

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 program 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.

2.1.2.3. Using Events to Monitor Memory Allocation

To illustrate events, we examine the problem of monitoring allocations and deallocations of memory, i.e., *malloc* and *free* in C terminology. A common programming error is to pass an invalid address for deallocation, especially one for a block that was previously freed. Here we show how errors in the use of the memory allocator can be detected.

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

event define 'free ('addr)
    event set-attribute 'addr $frees_arg
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
    push-block
    create-local $qindex
    create-local $free_addr
    set $qindex = $qlen('malloc)  # $qlen gives length of specified event queue
    set $free_addr = $value('free, 1, 'addr)
    while ($qindex > 0)
        if ($value('malloc, $qindex, 'addr) == $free_addr)
            event remove 'free, 1  # remove first token from queue for 'free
            event remove 'malloc, $qindex  # and $qindex'th token for 'malloc
            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
    pop-block
end
```

Figure 3. Memory allocator monitor using Dalek events.

Figure 3 shows the definition of two primitive events, 'malloc and 'free, which correspond to invocations of the allocation and deallocation routines. 'malloc records (using **event set-attribute**) in its attribute 'addr the address of the memory block allocated; 'free records in its attribute 'addr the address of the memory block to be deallocated. $quiet_finish() is a Dalek built-in function that resumes execution of the inferior process until it returns from the current procedure. $quiet_finish() returns the procedure's return value. $frees_arg is a global convenience variable set at the breakpoint inside *free* to the address of the block being freed (i.e., *free*'s argument). Since *free* is a system-dependent library routine, how $frees_arg is set is also system-dependent.

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 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 only by 'free. The matching of 'malloc and 'free events is performed using a simple while loop, which sequentially compares the attribute 'addr passed by 'free with those returned by the previous 'malloc events.

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 their calling functions, we can make the crucial transition from merely detecting the symptoms of a bug to discovering the location of its cause.

Searching the history list is also useful in different contexts. Suppose, for example, we detect that the contents of an allocated structure have been corrupted. At that point, the user can type in a while loop to search the history list to determine whether the address of that structure was ever freed.

2.1.3. Using Dalek to Measure Performance

Dalek can also serve as a tool for measuring run-time performance of the inferior process. In particular, it can be used to time parts of a program and to profile sections of code. A considerable advantage of using Dalek in these roles is that the part of the program on which information is to be gathered, as well as the precise information to be gathered, can be specified interactively and dynamically, without modifying, recompiling, and relinking the inferior program's source.

Dalek provides three built-in functions that return the cpu time used by the inferior process. $utime() returns the inferior's user time, $stime() returns the inferior's system time, and $ttime() returns the inferior's total 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; total time is the sum of user time and system time.) Unfortunately, only $utime() is accurate. $stime(), and therefore $ttime(), is not accurate since some of the cost of the debugger controlling the inferior process is charged as system time to the inferior process. These functions can be used in the code that is executed as part of a breakpoint or event, and therefore can easily be changed interactively and dynamically.

One use of Dalek's timing functions is for tuning a program. For example, suppose we want to determine the times for both successful searches and unsuccessful

searches of a symbol table. This information can give us feedback on the appropriateness of our data structure representation, perhaps suggesting that a hash table would perform better than a linked list. To obtain these times, one breakpoint can be set on entry to the search function and another breakpoint can be set right before the search function returns. The first breakpoint records the starting user time; the second adds the difference between the current user time and the recorded starting user time to the running sum for either successful searches or unsuccessful searches according to the value the search function is returning. Alternatively, the breakpoints could raise events that include the time information as attributes. That would allow a more detailed analysis, say to compute averages and variances, since each instance of an event is recorded on the event history list.

Gathering timing information using Dalek, instead of using tools like gprof(1) or including timing code in the source program, provides the user with more control over what is being measured and is not intrusive. As illustrated in part by the above example, individual or groups of procedures or statements can be measured, using arbitrary conditions to select those of interest. gprof does not provide such fine-grained control. Including timing code in the source program is viable, but that requires recompilation and can affect overall user time measurements. The number and kinds of changes to the source code required by that approach can also become unwieldy. For example, suppose we want to find the cumulative time spent executing in procedure P for those invocations from procedure Q. To do such measurements by modifying the source code requires either an additional parameter in procedure P's interface or a global variable; either of these changes is cumbersome in a program of any size. In Dalek, the name of the calling procedure N levels up in the stack frame can be determined using $func(N).

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. This kind of information is easily obtained by setting breakpoints at appropriate points in the code and specifying breakpoint commands to maintain convenience variables; **if** statements and functions are used for performing the more complicated measurements.

2.1.4. On the Fly Code Patching

Dalek can also be used for "code patching", in which original (e.g., buggy) program code can be effectively replaced, during debugging, with code written in Dalek. This kind of patching is temporary, affecting only how the inferior executes while under control of the debugger; it does not modify the program's executable file or the source code. Using such code patching, multiple errors can be found and fixed without exiting the debugger and recompiling the program. For example, suppose that debugging indicates that

```
symbol[end—start] = '\0';
```

which appears as line 38 in the file symtab.c should really be

```
symbol[end—start+1] = '\0';
```

Then, a breakpoint can be placed at that line whose (Dalek) code simply executes the correct statement and then jumps around the bad code. That is,

```
break symtab.c : 38
commands
   silent # suppress the default announcement of encountering a breakpoint
   set symbol[end–start+1] = '\0'
   jump 39 # skip over the bad statement; continue execution of the inferior
end
```

Each time the inferior reaches that line, the commands associated with this breakpoint are executed instead of the original, buggy code. Using techniques similar to the above, new (Dalek) code can be added to a program to initialize variables, etc. It is also easy to replace entire functions or groups of statements since the Dalek language provides conditional and looping statements and functions.

2.2. Design Decisions and Tradeoffs

Our main goal in designing Dalek was to focus on the fundamental problems in debugging and to offer solutions to those problems. One early decision we made was not to expend effort developing a graphical interface. While such an interface would be useful, we chose instead to work on what we consider to be the more basic problems.

Dalek is intended to work on existing executables, preferably compiled with the '-g' flag so that Dalek can obtain the executable's symbolic names, types, and line numbers. Furthermore, Dalek requires no modifications to the compiler or program's source code. In fact, Dalek encourages a very interactive, dynamic style of debugging, e.g., as illustrated above with the code patching and measurements examples. This approach allows the user to explore a program during its execution, adding routines as desired. 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 routines. Furthermore, seemingly minor modifications to the source code can substantially alter the symptoms exhibited by bugs, especially if pointers are involved.

The Dalek language provides all the mechanisms that programmers generally find useful: assignment statements, **if** and **while** statements, blocks, local 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. Furthermore, the same language is used at the debugger's command level and within code written for breakpoints and events. 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.

Providing **if** and **while** statements within the debugger language means we had to resolve language semantic and implementation issues for combinations of features. For example, single-stepping the inferior from within a while loop normally means that the next debugging command to be executed is the next one in the while loop, unless that step causes the inferior to execute another breakpoint. Thus, Dalek's implementation needs to maintain a stack of pending command blocks.

The notion of event-based debugging is common in the debugging of concurrent programs (e.g., [Baiardi83], [Bates82], [Bates83], [Bates88], [Elshoff88], and [Lin88]). However, it has not been applied to debugging of sequential programs. Our approach to event recognition differs considerably from other such approaches. Other event-based approaches typically use a special notation (e.g., based on pattern matching) to specify how lower-level events should be combined into higher-level ones. Our approach, on the other hand, uses dataflow and programmable nodes for that purpose. Our approach is intentionally low-level to give the user flexibility and to allow

us to explore the capabilities of our prototype. Unlike other approaches, it does not constrain event recognition by preconceived notions of how higher-level events will be formed. One drawback of our approach, however, is that even common patterns need to be explicitly programmed. We will remedy that, after gaining further experience with Dalek, by providing a macro facility or libraries.

One final interesting design decision we made was to provide "broadcast breakpoints": breakpoints set en masse at the entry to every procedure whose name matches a given regular expression. Such breakpoints can share a common block of breakpoint commands. Simple procedure tracing, for example, can be effected by issuing a single broadcast breakpoint command with "*" as the regular expression and

```
silent # suppress the default announcement of encountering a breakpoint
print $func(0) # output name of currently executing procedure
cont # continue execution of the inferior process
```

as the command block. The command block uses the built-in Dalek function $func() to obtain the name of the function associated with the current frame. Another form of broadcast breakpoints associates a breakpoint with each instruction in a given address range whose opcode matches the specified pattern.

2.3. Implementation Issues

Dalek's implementation is based on *gdb*'s. That, of course, saved us considerable time. However, the support for some of Dalek's high-level features required some innovation. Two particularly interesting implementation issues involve broadcast breakpoints and garbage collection.

2.3.1. Broadcast Breakpoints

Recall that a command establishing a broadcast breakpoint can specify many procedures at which to place a breakpoint. Execution of a program with a large number of breakpoints under *gdb*, however, can be very slow. Specifically, *gdb* removes *all* breakpoints from the inferior's code space when control passes from the inferior to the debugger; it re-inserts them when control passes back. Removing and re-inserting each breakpoint requires two calls to ptrace, each of which requires the UNIX kernel to perform two context switches.

Dalek avoids this cost unless the inferior actually executes a breakpoint. The one potential problem with this solution is that Dalek could display incorrect assembly code if it displays exactly the instructions it reads from the inferior's code space since those might include the breakpoint instructions. That would typically lead to "framing errors", causing subsequent instructions to be incorrectly displayed. Dalek avoids this problem by intelligent use of *gdb*'s existing breakpoint data structures to show the original code (i.e., without the breakpoint instructions).

2.3.2. Garbage Collection

gdb uses a very simple garbage collection algorithm. As it executes each command, it links together on a "freeable" list all memory that it allocates. If the contents of a node might be needed later (e.g., when a new convenience variable is assigned) it removes the node from the freeable list. After executing each command, *gdb* frees all nodes on the freeable list.

Dalek's high-level debugging features, however, require a more complicated garbage collection scheme. As one example, Dalek's functions can return values that can be used in expressions. Their values cannot be freed, as *gdb* normally would, until after they are used in the evaluation of the rest of their enclosing expression. Our implementation, therefore, delays freeing an object until the command that created it has been exited with no possibility of resumption. In essence, we maintain the freeable

list as a stack.

3. Evaluation

3.1. Implementation Effort

We have implemented all of the Dalek features described in this paper and several other features. (See [Crawford90] for more details on Dalek's implementation.) Our implementation augments *gdb*'s (version 3.0) approximate 35,800 lines of C source code by roughly 9,400 lines. Development activities took about one person-year.

The major effort in implementing Dalek was spent

- developing support for **if** and **while** statements, procedures, and blocks with local variables;
- developing support for events;
- and getting these features to work well together and with the rest of *gdb*'s features.

Many of the required changes were straightforward. The changes for the **if** and **while** statements, for example, required adding new routines for interpreting those statements. Other implementation changes, such as dealing with stacks of pending commands and needing a better garbage collection algorithm mentioned in section 2.3, required more implementation effort. The part of Dalek's implementation that deals with events and the dataflow engine is fairly independent of the rest of the implementation. This part did, however, require several "hooks" into the scanner so that names of attributes, for example, were recognized as such.

One desirable feature that has not been implemented is to allow new types to be defined within Dalek. That is, convenience variables are restricted to have types that have been declared in the inferior process.

3.2. Performance Data

The memory allocation example presented in section 2.1.2.3 was actually used to find an insidious memory allocation bug in Dalek itself. That is, we used Dalek 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 than manually checking the arguments in each invocation of *malloc* and *free*. The two major cost factors in this example are in context switching between Dalek and the inferior process (two for each ptrace system call) and in interpreting Dalek commands.

Dalek's implementation is currently far from optimal. For example, each command inside a while loop is parsed from its source text on each iteration of the loop. This behavior fits better with *gdb*'s existing structure. It also affects the performance of Dalek's event handling since commands in event handling code are interpreted in the same manner.

3.3. What would we do differently?

Dalek was developed as a prototype to allow us to explore our ideas in debugging. As such, we decided that it would be easier to extend an existing debugger than it would be to develop an entirely new debugger from scratch. We chose to use *gdb* as

our starting point because its source is readily available, it runs on a wide range of architectures, and its command language is fairly clean both syntactically and semantically. While *gdb* gave us a very good platform on which to build, it also caused us some problems, for example, due to the integration of noncompatible features.

One such problem is that *gdb* parses its input in a line-oriented manner. That constrained our syntactic options in designing the Dalek language and explains why we have commands like **push-block**, **pop-block**, and **create-local** to effect a local scope instead of more standard (and palatable) syntax. Also, as mentioned above, commands are parsed each time they are encountered, which has an adverse affect on Dalek's performance. Translating the commands into an internal representation would considerably improve Dalek's performance.

A second problem we had building Dalek on top of *gdb* is related to breakpoints. When single-stepping under *gdb* on some architectures (e.g., Sun 3's), if a breakpoint instruction is encountered, the breakpoint is not detected and the associated commands are not executed. That behavior may be reasonable in *gdb* but is problematic in Dalek since the breakpoint might be associated with raising an event or used to transfer control to a code patch.

Ideally, we would have started the language and implementation from scratch. In fact, we are considering a new version of Dalek whose implementation would continue to use *gdb*'s low-level routines for managing the inferior, but would use an entirely new command interpreter that would support free form input and parse commands into an internal representation, as suggested above.

One item on our wishlist is additional operating system and architectural support for debugging. For example, the only standard UNIX system call to support debugging is ptrace. As mentioned earlier, ptrace is very primitive and expensive, requiring two context switches for each word of data transferred between the debugger and the inferior on some machines (e.g., a VAX). The kind of high-level debugging encouraged by Dalek can require a large amount of information to be passed between the debugger and the inferior. For example, event attributes are often values of variables in the inferior, which need to be copied into Dalek's space when an event is raised. Additional support for debugging could speed up the high-level debugging we describe here. For example, allowing one process (the debugger) to read directly the address space of another (the inferior) can greatly reduce the number of system calls. That kind of facility has been implemented by treating processes as files [Killian84]; e.g., System V Release 4 UNIX systems provide the "/proc" file system. Another approach is to allow the debugger to share some of the inferior process's address space [Aral88].

4. Conclusion

This work is significant for two reasons. First, it defines mechanisms for debugging at a higher level than existing debuggers. In particular, Dalek's debugging language provides: a full repertoire of conventional programming features (e.g., local variables, conditional and looping statements, and procedures); ways to define, raise, and recognize events; and ways to subsequently search and correlate events in the "event database", i.e., the event history list. The examples demonstrated the utility of these mechanisms in solving real debugging problems. Second, this work demonstrates that those mechanisms are feasible by incorporating them into a working prototype, i.e., Dalek. Having the prototype has given us an opportunity to obtain valuable feedback on our ideas and their implementation, and a powerful debugging tool that we and others find most useful. Having the prototype has also encouraged us to experiment with Dalek's use as a dynamic performance measurement tool, as described earlier. Dalek is being incorporated into a future release of *gdb*.

Dalek's execution time for high-level debugging is in general reasonable, and certainly much less than would be required for a user to do the same work manually. For example, for a given problem that would require several person-hours employing other debuggers, Dalek might require minutes of computer time. In some cases, however, Dalek's execution time can be high. As mentioned earlier, additional operating system and architectural support for debugging would drastically reduce costs in those cases.

Our future work will include working on Dalek language mechanisms, studying further ways to optimize Dalek's implementation, and generalizing our approach to allow debugging of concurrent programs.

Acknowledgements

Many thanks are due to Richard Stallman and the Free Software Foundation for making *gdb* available to the public. Having such a good base on which to build Dalek greatly expedited our work. Jim Kingdon of FSF is integrating Dalek back into *gdb*. Mudita Jain, Carole McNamee, Chris Wee, and Cui Zhang provided very useful comments on Dalek and on earlier drafts of this paper. Early work on this project was supported in part by System Integrators, Incorporated (SII) and the State of California under the MICRO program.

References

[Aral88] Z. Aral and I. Gertner. "Non-Intrusive and Interactive Profiling in Parasight". *Proc. of the ACM/SIGPLAN Parallel Programming: Experience with Applications, Languages and Systems*, 21-30, July 1988.

[Baiardi83] 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, 98-106, March 1983.

[Bates82] P. Bates and J.C. Wileden. "EDL: A basis for distributed system debugging tools". *Proc. Fifteenth Hawaii International Conference on System Sciences*, Honolulu, HI, 86-93, 1982.

[Bates83] 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, 107-111, March 1983.

[Bates88] P. Bates. "Debugging heterogeneous distributed systems using event-based models of behavior". *Proc. Workshop on Parallel and Distributed Debugging*, Madison, WI, 11-22, May 1988.

[Crawford89] R.H. Crawford, W.W. Ho, and R.A. Olsson. A Dataflow Approach to Event-Based Debugging. CSE-89-7, Div. of Computer Science, University of California at Davis, May 1989.

[Crawford90] 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.

[DBX83] "DBX (1)" in *UNIX Programmer's Manual*, 4.2 Berkeley System Distribution, Vol. 1, Computer Science Division, University of California, Berkeley, CA, August 1983.

[Elshoff88] I.J.P. Elshoff. "A distributed debugger for Amoeba". *Proc. Workshop on Parallel and Distributed Debugging*, Madison, WI, 1-10, May 1988.

[Johnson78] 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.

[Katseff79] H. P. Katseff. "Sdb: a symbolic debugger". in *UNIX Programmer's Manual, 7th Edition, Vol. 2C* Bell Laboratories, Holmdel, NJ, January 1979.

[Killian84] T. J. Killian. "Processes as files", Proc. of the Summer 1984 Usenix Conference, 203-207, June 1984.

[Lin88] C. Lin and R.J. LeBlanc. "Event-based debugging of object/action programs". *Proc. Workshop on Parallel and Distributed Debugging,* Madison, WI, 23-34, May 1988.

[Maranzano79] 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.

[Stallman89] R.M. Stallman. *GDB Manual (The GNU Source-Level Debugger), Third Edition, GDB version 3.1,* Free Software Foundation, Cambridge, MA, January 1989.

[Tulloch83] J. Tulloch and M. Alvarado. *Doctor Who—The Unfolding Text,* Macmillan Press, 1983.

AUTHOR INFORMATION

Ron Olsson received the B.A. degree in mathematics and in computer science, and the M.A. degree in mathematics, from the State University of New York, College at Potsdam. He received the M.S. degree in computer science from Cornell University and the Ph.D. degree in computer science from The University of Arizona. Dr. Olsson has been assistant professor of computer science at the University of California at Davis since 1986. His research activities center around concurrent programming languages—focusing on issues in design, implementation, optimization, and debugging—and system software.

Rick Crawford earned his B.S. in Mechanical Engineering from the University of California at Berkeley in 1979. After receiving firsthand experience in the "real world" regarding how large software projects should not be managed, he sought sanctuary amidst the relative sanity of academia. He is currently a graduate student in computer science at the University of California at Davis. His programs (except for Dalek) are always bug free.

Wilson Ho is a graduate student and Ph.D. candidate in the Department of Electrical Engineering and Computer Science at the University of California at Davis. He received his B.S. in computer science from the University of Hong Kong in 1986, and his M.S. in computer science from the University of California at Davis. His research interests include debugging of distributed programs, distributed file systems, and programming environments. His programs (except for Dalek) are never bug free.