Sequential Debugging at a High Level of Abstraction

RONALD A. OLSSON, RICHARD H. CRAWFORD, W. WILSON HO, and CHRISTOPHER E. WEE
University of California at Davis

Debugging seems to focus almost exclusively on the debugging of concurrent programs and on graphical interfaces for debuggers. While these areas are very important, the emphasis on them tacitly assumes that the problems of debugging sequential programs have been solved. We disagree. We find state-of-the-art sequential debuggers to be lacking, so we have developed our own sequential debugger to demonstrate that significant further progress can be achieved.

A trend over the years has been for sequential debuggers to provide mechanisms that let users interact with their programs at higher levels of abstraction, closer to the abstractions in their programs. Debugging tools have evolved from programs that let you examine contents of memory locations to modern, source-level debuggers that let you interact with your program at the source-code level, since they understand abstractions like variables, procedures, and statements. Fundamental to such debuggers is the concept of a breakpoint, a programmer-defined location in the code at which execution is to be suspended so you can examine variables or so the debugger can execute some debugging code.

Some source-level debuggers, like Sun Microsystems’ DBX tool and the Saber C and Turbo debuggers, provide graphical interfaces to simplify how you interact with the debugger and to help you visualize program behavior. Unfortunately, these widely used debuggers do not go far enough: They still require you to interact at a very detailed level. These debuggers typically have two significant inadequacies:

* They provide only very limited control over the actions taken when a break-
point occurs. Often, they provide only conditional execution of the entire block of commands associated with a breakpoint. More powerful and discriminating means of control are desirable, like a loop to print out the contents of a linked list or a loop to single-step the code being debugged until one variable is greater than another. (The Digital Equipment Corp. VAX/VMS Debug and the Saber C debuggers do provide some such features.)

- They provide no way of conveniently or safely correlating the occurrence of several logically related breakpoints into a single, more abstract occurrence. This capability is desirable because abstractions in a program often require a series of procedure calls. You can easily be overwhelmed by all the information output by a debugger and might fail to realize the significance, when taken together, of two pieces of information that a debugger prints at different times.

Our debugger, Dalek, remedies both inadequacies. (The name comes from the semi-mechanical alien life form in the British television series Doctor Who that tried to exterminate everything in sight.) Dalek's language provides conditional and looping statements, blocks, local variables, procedures, and functions. (The box below defines some Dalek tokens used in this article.) This approach is similar to that taken by Mark Johnson, 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 code execution --- as a way to form higher level abstractions during execution.

Dalek uses 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.

OVERVIEW

Dalek is similar to many other modern debuggers in its overall operation. Dalek and the code being debugged execute as separate processes, with Dalek controlling, observing, and altering the code's execution according to code written in the Dalek language. You can specify Dalek code dynamically during a debugging session, so you can specify what is of interest as you glean information during execution. Also, such code can be read from a file. Dalek code can be executed from the debugger's command level or automatically at a breakpoint or when an event is triggered. Dalek operates on compiled C programs; the source code is not modified, recompiled, or relinked as a result of debugging.

Dalek is based on GDB, the GNU Software Foundation's Gnu Project's debugger. It enhances GDB's functionality in two significant ways.

First, Dalek extends the original GDB language with mechanisms that make it more like a general-purpose language. Because the new mechanisms are completely integrated with the standard debugging features, the Dalek language is fully programmable. The new mechanisms supported by Dalek include

- a conditional statement,
- a looping construct,
- blocks with local convenience variables (a convenience variable is a variable that the user defines and manipulates within the debugger; GDB provides only global convenience variables), and
- procedures and functions.

Second, Dalek provides ways to define, raise, recognize, and combine events.

To illustrate the Dalek language, consider the Dalek code in Figure 1. The code defines a \texttt{Sprint\_list()} procedure, which can be invoked interactively from Dalek's command level or from any point during code execution by simply setting a break-
point whose commands include a call to the procedure. As a simple example, Sprint_list() can be invoked from a break-
point at line 27 of the file Goog.c by the commands

break Goog.c: 27
commands
silent
call Sprint_list(function, head)
end

Silent suppresses the default announce-
ments of encountering a breakpoint, the Call statement calls the Sprint_list procedure, passing Symth_head as an argu-
ment, and Cont continues execution of the program being debugged.

We could have written the body of Sprint_list() in-line as commands, but we wrote it as a function so it can be invoked
directly from the command level or from another breakpoint, if desired.

Similar to the Sprint_list() example, you can write Dalek code to ensure that the code being debugged maintains certain invariants
about its data structures. For example, you could invoke code to check that a circularly
linked list is indeed circular on entry to and
exit from each procedure that manipulates
the list, or even at every step of the code. This
approach is more flexible than using compiled-in assertion checks (like the Assert
macro in C), since you can decide what in-
variants to check and when to check them as
you find it necessary during actual debug-
going. You can save the invariant-checking
code for use in later debugging sessions.

To see further how you can use Dalek to
tack a debugging problem, consider a pro-
gram containing a module that main-
tains a sorted linked list. This sorted list is
kept local to the module and cannot be
accessed directly anywhere outside this
module's scope. All operations on the list
are performed through a few explicitly ex-
ported functions. When this module is
combined with other parts of the program
and the program is executed, the contents
of this list are somehow corrupted. Careful
examination of the list module leads you to
believe that it is not the source of the cor-
rupation. Therefore, some other part of the
program must contain erroneous code
that has overwritten this linked list.

This kind of bug is common in pro-
grams written in languages like C that
allow the use of pointers. Most likely, it is
caused by a pointer incorrectly referencing
a memory location occupied by the linked list.

The debugging problem is to locate the erroneous code, since it could be in any
part of the program. To narrow the bug's possible location, you can trace, step by
step, the program's execution.

Figure 2 shows Dalek code that automa-
tes the program tracing. The function
Slist_sorted() verifies the validity of the
sorted list and returns 0 when the list is no
longer sorted. A breakpoint is set in the
function Initialize_sorted_list(), where
the sorted list is first used. The command
block associated with this breakpoint be-
gins with a loop that single-steps execution
of the program, stopping when the list is
first found to be corrupted. It then prints a
trace of the current call stack. (We con-
sider this kind of debugging to be the last
resort—you use this technique after fail-
ing with the more standard ones.)

The key to success here is that Dalek
allows single-stepping within While loops.
You could also use hardware data
watchpoints, available in some architec-
tures, to help solve this kind of debugging
problem. They let specified memory cells be
monitored, generating an interrupt when
such a cell is read or written. However, some
way of filtering through the enormous num-
ber of generated interrupts is needed.

EVENTS

You can define an event in Dalek to
capture the occurrence of any interesting
activity in the code execution. To define
an event, you declare names for it and
for its attributes and write a block of Dalek code. How does Dalek know when to execute an event's code? The simplest way is to explicitly raise that event by name. We call events activated this way primitive events, and they are typically raised by an event-raise command executed at a breakpoint. The code associated with an event will then try to recognize a valid occurrence of that event. The attributes associated with an event contain information meant to characterize a particular occurrence of the event. The Dalek code you write for an event's definition can assign values to its attributes from variables in the original code, debugger-convenience variables, or computation based on those variables.

Dalek also supports high-level events. When defining a high-level event, you must specify the names of other events on which it depends. A high-level event is not explicitly raised; instead, Dalek can automatically execute a high-level event's code whenever an occurrence of a primitive event on which it depends is successfully recognized.

Dalek provides a coarse-grained dataflow view of events. 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 whose activation and subsequent behavior depend on those primitive events. Thus, the structure of the dataflow graph is implicit in the events you define. Dalek derives it automatically from the dependencies you explicitly specify between primitive and higher level events.

Activity in the code being debugged—like reaching a breakpoint—can raise a primitive event, which if recognized will generate a token. This token carries copies of the attributes characterizing that instance of the primitive event, letting that occurrence be distinguished from other occurrences of the same primitive event. The tokens generated by recognizing a valid instance of a primitive event flow to any high-level events that depend on the primitive event. The arrival of such a token at a high-level event can trigger that high-level event, which in turn can generate tokens that trigger yet higher level events. Figure 3 shows a dataflow graph that summarizes Dalek's approach to event recognition.

When a high-level node in Dalek's event-dataflow graph is triggered, it executes its own Dalek code to decide whether to recognize a valid instance of that high-level event. In making that decision, the code associated with a high-level event has access to the tokens (and their attribute values) from all occurrences of the lower level constituent events on which the high-level event depends. The tokens from a high-level event's lower level constituents are stored on its own private queues and are accessed with predefined Dalek functions.

If the Dalek code associated with an event decides that it should recognize an instance of that event, it assigns values to its own attributes, as we described earlier. A high-level event then generally removes from its incoming queues the tokens on which the event's code based its decision to avoid repeated recognition of the same tokens when it is triggered later. When the high-level event's code finishes execution, tokens embodying the new event instance propagate to all higher level events that depend on it, and the recognition phase begins anew.

If the high-level event's code decides that an appropriate combination of tokens from constituent events does not exist on its incoming queues or that their attributes fail to satisfy the desired relationships, the code can execute Dalek's Don't-propagate command. This suppresses the normal generation of tokens representing this event and the subsequent passing of those tokens to any higher level events that depend on it.
An event history records all recognized event occurrences and their attributes. It may be browsed selectively by the user in interactive mode or accessed programmatically via predefined functions in the Dalek language.

To demonstrate events concretely, Figure 4 shows a spreadsheet program and Dalek debugging code. It has two bound primitive events (raised at breakpoints) that are combined into a higher level event. Figure 5 shows another example of how you can use events. In this example, the primitive events are free-floating — unlike the primitive events in Figure 4, the primitive events in Figure 5 are raised during execution of Dalek code that is not bound to any particular breakpoint.

**EXAMPLE APPLICATIONS**

Dalek's debugging language makes it suitable for many applications, three of which we present here: an alternative to code patching, procedure tracing, and performance measurement and code profiling. These applications further show the versatility and power provided by Dalek's full programmability and its event mechanisms. Although some people might not consider these applications to fall in the realm of traditional debugging, we have found that the interpretive-like environment Dalek provides for compiled programs encourages the integration of previously separate activities (program testing is another example).

*Alternative to code patching.* You can use

```c
void RowChanged (int row) {
    int column;
    for (column=0; column < LastColumn; column++)
        UpdateCell (row, column);
}

void ColumnChanged (int column) {
    int row;
    for (row = 0; row < LastRow; row++)
        UpdateCell (row, column);
}

void UpdateCell (x, y) {...}
```

```c
# Define a breakpoint at procedure entry and
# associate a primitive event with that breakpoint.
break RowChanged
    commands silent
    event raise 'RowChanged (row)
        cont
    end
    event define 'RowChanged(row)
end

# Define a breakpoint at procedure entry and
# associate a primitive event with that breakpoint.
break UpdateCell
    commands silent
    event raise 'UpdateCell (x, y)
        cont
    end
    event define 'UpdateCell(x, y)
end

# Define a higher level event. This event is "raised" by
# the primitive events defined above.
break RowThenCell(x, y) = RowChanged + UpdateCell
    if (0 = $len('RowChanged'))
        event dont-propagate
    end
    event define 'RowThenCell(x, y)
end
```

Figure 4. Spreadsheet example: (A) Excerpt from a spreadsheet program and (B) Dalek commands and definitions. In the example, both RowChanged() and ColumnChanged() call UpdateCell(). Suppose you are interested only in execution paths that run through RowChanged() and then UpdateCell(). You would use Dalek's dataflow mechanism to filter out all UpdateCell() invocations that do not first invoke RowChanged().
We assume each successfully recognized primitive event was
a successful insert or delete (it changed the list's length).

```
# event define 'insert' (value)
end

# event define 'delete' (value)
end
```

# Assumes program being debugged
# has one pointer (listptr)
# that both inserts and deletes
# occur only at this pointer.

```
set last = listptr
while (running)
  step
  if (last != listptr)
    if (last == 0 & & listptr)
      event raise 'insert' (listptr->value)
      insert into empty list
    else if (last & & listptr == 0)
      event raise 'delete' (last->value)
      delete, making empty list
    else if (last & & listptr->next == last)
      event raise 'insert' (listptr->value)
      insert into existing list
    else if (last & & listptr->next != last)
      event raise 'delete' (last->value)
      delete from existing list
  endif
  set last = listptr
endwhile
```

Figure 5. Free-floating events.

Dalek to effectively replace code in the program being debugged with code written in Dalek. This alternative to patching machine code has a temporary effect, affecting only how the code being debugged executes while under control of the debugger: it does not modify the program's executable file or the source code. We have used this kind of patching to find and fix multiple errors without exiting the debugger and recompling the source code.

Although you can use this technique with many other debuggers, it is easier to use it in Dalek because Dalek's language provides conditional and looping statements and functions. These let you easily replace groups of statements or entire functions, or add new code to, for example, initialize variables.

The basic technique is to set a breakpoint at the start of the faulty code. The Dalek code associated with the breakpoint can then perform the desired action, and, if necessary, can execute a special jump command to skip over code that should no longer be executed.

**Procedure tracing.** Dalek does not provide a separate trace command; instead, it provides a more general mechanism — broadcast breakpoints — that, with its full programmability, makes getting a trace of selected procedure calls simple as a special case. Broadcasting lets you give a single command to establish an ordinary breakpoint at the entry to every procedure whose name matches a regular expression, and broadcasting lets you associate a common block of debugging commands with those breakpoints. Without a broadcast-breakpoint facility, you would have to individually establish a breakpoint and give the associated breakpoint code at each procedure — a tedious activity.

Thus, to have Dalek print out the names of all procedures entered during program execution, you just specify a single broadcast-breakpoint command with the wild card * as its regular expression and the following code as its command block:

```
if print "%s\n", $func(0)
```

The predefined Dalek function $func(n) returns the name of the function in the code being debugged that is associated with the nth frame of the procedure-call stack. When n equals 0, in this case, it returns the name of the procedure in which the code being debugged is executing.

More selective tracing is also possible by specifying a more restrictive regular expression or by refining the code in the broadcast breakpoint's command block. Two examples of the latter include

- printing only when a certain procedure appears somewhere on the call stack (the call stack can be searched with a While loop) and
- printing only when the value of some convenience variable satisfies an arbitrary Boolean expression (the value of this variable might be conditionally modified by the code associated with events or other breakpoints).

We have used broadcast breakpoints to get information used in correlating the activities by which programs access their runtime (procedure) stacks. This information is important in assessing the utility of certain architectural features. The conditional probabilities we needed to measure were a push of a stack frame followed by the push of another stack frame, a push followed by a pop, a pop followed by a push, and a pop followed by another pop. Figure 6 shows how we programmed Dalek to gather and compute such information.

We have also used broadcast break-
points to solve debugging problems like the one in Figure 2. Broadcast breakpoints are faster than the single-stepping used in Figure 2, but their granularity is coarse, since they occur only when a procedure is entered. Using them will give you only the name of the routine invoked after the routine in which the contents of the list were corrupted, rather than the source statement actually causing the corruption. Thus, the fastest way to zero in on such a bug is for the broadcast-breakpoint commands to increment a counter to determine how many times procedures are invoked, return the code until one procedure call before that state, and only then initiate the While loop that single-steps.

Performance measurement and code profiling.
You can use Dalek to time parts of a program and to profile sections of code. A considerable advantage in using Dalek for these tasks is that you can interactively and dynamically specify the part of the program for which information is to be gathered, as well as the precise information to be gathered — without modifying, recompiling, and relinking the source code.

Dalek provides three predefined functions that return the CPU time used by the process in the code being debugged, corresponding to how Unix reports CPU times. The Dalek function Sutime() returns the user time, Sstime() returns the system time, and Stotal() returns the total time. User time is the time the CPU spends executing the instructions in the process's address space, while system time is the time the CPU spends executing system calls invoked by the code.

Unfortunately, only Sutime() is accurate. Sstime(), and therefore Stotal(), is not accurate because some of the cost of the debugger controlling the process being debugged is charged as system time to the debugger. And even Sutime() is not always accurate on all machines. For example, on machines that use instruction pipelining, Sutime() 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 process being debugged and the debugger.

Still, you can use these timing functions in the debugger code that is executed as part of a breakpoint or event, and therefore interactively change what parts of the code being debugged are monitored.

One use of Dalek's timing functions is to tune a program's performance. For example, suppose you want to determine the times for both successful searches and unsuccessful searches of a symbol table. This information can give you feedback on the appropriateness of your data-structure representation, perhaps suggesting that a hash table would perform better than a linked list.

To get these times, you can set one breakpoint on entry to the search function and another breakpoint right before the

```plaintext
# Broadcast breakpoints. The commands compute conditional probabilities.
# Initialize some Dalek variables. Theses variables are global in scope.
set Sutime = "none"
set Spush_pop + Spop_pop = Spop + Spop = Spush_pop = 0
set Srunning = 1

proc-break ->
  silent
  # Execute for every procedure entry
  if (Slast = "push")
    set Spush_pop ++
  else if (Slast = "pop")
    set Spop_pop ++
  endif
  set Slast = "push"
  if ("_exit" = $funcname) =
    set Srunning = 0
  else if (0 == Stop_frame())
    cont
  else
    call Squirt_finish()
    if (Slast = "pop")
      set Spop_pop ++
    else if (Slast = "push")
      set Spush_pop ++
    endif
    set Slast = "pop"
  endif
  end

# Start execution of the program:
run while (Srunning)
  cont
endwhile

# Print one result:
print "Conditional probability that PUSH is followed by a PUSH = " / ($push / (push + pop - push))
```

Figure 6. Computing conditional probabilities for stack usage.
# After-the-fact use of history list to derive
# new information: the maximum
# length reached by a linked list.

# Run the program, Dalek records
# interesting events.
# Consult Dalek's history list of interesting events
# and calculate maximum length of the list.
print "%d\n", $max_length()

(A)

# Using Dalek's history mechanism, you walk through the execution history
# of the program and
# examine the occurrences of two important events -- 'insert' and 'delete', thus
# letting you calculate
# the maximum length of the linked list.
function $max_length()

    local $len, $max_len, $ins_q_index, $del_q_index
    set $len = $max_len = 0
    set $ins_q_index = $del_q_index = 1

    while ($ins_q_index <= $len (insert) && $del_q_index <= $len (delete))
        if ($time (insert, $ins_q_index) < $time (delete, $del_q_index))
            set $len++
            set $ins_q_index++
        else
            set $len--
            set $del_q_index++
        endif
    if ($len > $max_len)
        set $max_len = $len
    endif

endwhile

# There may have been a tail of inserts with no more deletes, so:
set $len = $len (insert) - $ins_q_index + 1
if ($len > $max_len)
    set $max_len = $len
endif
func-return $max_len
)

end

(B)

Figure 7. (A) Using the event-history list to examine program behavior: The free-floating events for 'insert' and 'delete' are defined and the attributes set as in Figure 5. (B) The definition of the $max_length function used by the event-history examination routine.

search function returns. The first breakpoint records the starting user time; the second adds the difference between the current user time and the 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, for example, to compute averages and variances, since each instance of an event is recorded on the event history.

Gathering timing information with Dalek — instead of using standard profiling tools like Unix's Gprof tool — or including timing code in the source program gives you more control over what is being measured, plus it is not intrusive. Profilers do not provide control as fine-grained as Dalek's timing functions. While including timing code in the source program is viable, doing so 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 you want to find the cumulative time spent executing in procedure P but only when it is invoked directly or indirectly from procedure Q. To do such measurements by modifying the source code would require either an additional parameter in P's interface or a global variable; either of these changes would be cumbersome in a program of any size. By contrast, Dalek code to do such measurements simply uses the $func predefined function described earlier to determine if Q is on the procedure-call stack.

You can also use Dalek as a simple codeprofiler and to gather other useful statistics about program behavior. For example, we have used Dalek to count how often a procedure or statement is executed, to find the maximum length a linked list reaches during execution, and to measure runtime-stack frame behavior. This kind of information is easily obtained by setting breakpoints at appropriate points in the code and specifying breakpoint commands to maintain convenience variables; you would use control statements and functions for the more complicated measurements.

You can analyze more complicated program behavior by using the event-history feature. Figure 7 shows how you can use the event history to determine the maximum length a linked list reached during execution, without having to rerun the code being debugged.

DEBUGGING STYLE

Dalek encourages a very interactive, dynamic style of debugging, as the examples in this article show. Dalek's full programability provides the benefits of a debugger fully integrated with an interpretive environment, despite the fact that it operates on a compiled source program.
You can write Dalek code when you find it necessary, like when execution of the code being debugged is suspended at a breakpoint in the source code or even when suspended at a breakpoint somewhere in the Dalek code.

This approach lets you explore a program during its execution, adding debugging code as desired by defining a new high-level event or a new debugging function, like `Slist_sorted()` in Figure 2.

Often in debugging, you have ardously brought the code being debugged to a critical point in its execution under the debugger’s control before you discover the need for such debugging code. Without full programmability like Dalek’s, you would have to limp along without these capabilities or to inject debugging statements (like `Assert statements`) into the source code, perhaps with conditional compilation. Not only would this require you to terminate the debugging session, modify the source code, and recompile and relink the executable file, but making such seemingly minor modifications to the source code could substantially alter the symptoms exhibited by bugs, especially if pointers are involved.

One drawback of the kind of high-level debugging encouraged by Dalek is that it requires a fair amount of interaction between the debugger process and the process being debugged. Unfortunately, that kind of interaction is generally expensive because it requires system calls and their concomitant context switches. But although such high-level debugging might require a few minutes of computer time, it can save you hours of your time.

Another potential drawback of any approach to debugging is that the debugger code may itself contain bugs. Dalek code is as susceptible to bugs as any other debugging code, perhaps more so since it supports and encourages the use of more complex features. To aid the debugging of the debugging code, Dalek provides a verbose mode in which Dalek displays each command as it is interpreted. Dalek also provides several ways to pause its execution as it interprets the debugging code: It lets you single-step through and set breakpoints in your debugging code.

**OTHER APPROACHES**

Our approach to event recognition differs considerably from other approaches, including those used by several debuggers for concurrent programs that support various forms of events. We use a dataflow graph with programmable nodes to specify how lower level events should be combined into higher level ones. Thus, event recognition in Dalek is an active process. Other event-based approaches typically use a special notation (often based on pattern matching) to specify event recognition. Dalek’s notion of persistent, user-definable attributes to characterize each occurrence of an event is also uncommon.

Our approach to event recognition is intentionally low-level to provide maximum flexibility. Unlike the other approaches, it does not constrain event recognition by preconceived notions of how higher level events will be formed. But one drawback of our approach is that even common patterns must be explicitly programmed. We will remedy that, after gaining further experience with Dalek, perhaps by extending the Dalek language to include higher level mechanisms or by providing a macro facility or event libraries. We anticipate the resulting language will retain most of its current low-level mechanisms.

A few sequential debuggers have used the event-based model or a similar one. For example, the notion of event associations in Snobol-4 lets event recognition be an active process, but it is not possible to define high-level events or to maintain an event history. Another approach using generalized path expressions lets you define variants of Dalek’s high-level events, but its event recognition is not active and it does not provide any feature like Dalek’s notion of persistent attributes. Also, that approach is not fully programmable. Being effectively limited to global integer convenience variables, this deficiency limits the type of information that can be communicated between path expressions. By limiting the event history to storing only counts of event occurrences (in contrast to recording the time stamp as well as the attributes for each occurrence), the practical value of that database is severely constrained.

**REFERENCES**


IEEE SOFTWARE
ACKNOWLEDGMENTS

Our work was aided greatly by hosting Dalek on GDB. We thank Richard Stallman and the Free Software Foundation for making GDB available to the public. We also thank Carole McNair and the anonymous referees for their useful comments on earlier drafts of this article. Our early work on Dalek was supported in part by System Integrators and the State of California under the Micro program.

Ronald A. Olson is an assistant professor of computer science at the University of California at Davis. His research interests include systems software and concurrent programming languages — focusing on issues in design, implementation, optimization, and debugging.

Olson received a BA in both mathematics and computer science and an MA in mathematics from the State University of New York at Potsdam, an MS in computer science from Cornell University, and a PhD in computer science from the University of Arizona.

Richard H. Crawford is a postgraduate researcher in computer science at the University of California at Davis. His research interests include debugging, programming environments, computer security, coding theory, self-replication and fixed points, and modal and other non-standard logics.

Crawford received a BS in mechanical engineering from the University of California at Berkeley and a MS in computer science at the University of California at Davis.

W. Wilson Ho is a graduate student and PhD candidate in computer science at the University of California at Davis. His research interests include debugging of distributed programs, distributed file systems, and programming environments.

He received a BS in computer science from the University of Hong Kong and an MS in computer science from the University of California at Davis.

Christopher E. Wee is a graduate student in computer science at the University of California at Davis. His research interests include real-time programming, debugging of real-time programs, and object-oriented programming.

He received a BS in computer science and engineering from the University of California at Davis.

Address questions about this article to the authors at Computer Science Div., University of California, Davis, CA 95616-8562; Internet debug@ecs.ucdavis.edu.

AdaMAT™ REDUCE THE LIFE CYCLE COST OF YOUR ADA PROJECTS.

AdaMAT DELIVERS VISIBILITY INTO THE QUALITY OF ADA SOURCE CODE.

AdaMAT automates the collection, analysis, and reporting of metrics for compilable Ada source code, including Ada as a Program Design Language (PDL). Therefore, the benefit of measurement is not out-weighed by the cost of collection. Quantitative reports assist you in detecting, isolating, and correcting quality problems that impact the reliability, maintainability, and portability of your Ada source code.

AdaMAT will help to minimize schedule slips and cost overruns resulting from poor quality software by identifying the kinds of problems your Ada code is likely to present during execution, maintenance, and porting.

AdaMAT has enjoyed wide use among all branches of the military. More and more, AdaMAT is being specified by name to address quality requirements of the DoD. Throughout the project life cycle — from design to deployment and beyond, AdaMAT is the QUALITY ANALYSIS TOOL OF CHOICE.

800-522-7321

Build Quality Into Your Ada Development.

Dynamics Research Corporation, 60 Frontage Road, Andover, MA 01810

MAY 1991