CHAPTER 3

A User's View of Dalek: Events

This chapter elaborates the view of events presented by Dalek to the user. Several examples with nontrivial Dalek language code are included for concreteness. The reader may find it helpful periodically to refer to the Appendix, which contains a quick reference guide to Dalek's syntax. An index to Dalek's commands and also to its predefined debugger functions appears at the beginning of the Appendix.

3.1. Overview of Events in Dalek

Events in Dalek can encompass virtually any interesting activity in the inferior process that the user can specify by procedural methods.[1] An arbitrary number of dynamically-typed attributes can be associated with an event. Every *occurrence* of an event can have its own attributes, in much the same way that every invocation of a procedure can have its own actual parameters. The analogy extends further: The definition of a *primitive* event specifies a list of its attributes' names and also a command block written in the Dalek language. However, in contrast to a procedure invocation with its actual parameters, an event occurrence, represented by a token containing the values of its attributes along with a timestamp, can linger indefinitely after the event's command block finishes "executing". 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 upon which the high-level

---

[1] Although realtime constraints can be *expressed* procedurally, most UNIX implementations thwart prompt transmittal of the results of realtime measurements. For this and other reasons, it is not practical to use Dalek's events for characterizing realtime activity.

event's own activation may depend and whose attributes that high-level event can access. Event definitions can be given interactively or read from a file.

For Dalek, we devised a novel, dataflow model of event recognition. In most models of dataflow, tokens flow between the nodes of a directed graph. Computation occurs at a node if it is activated by the arrival of a token, which carries operand data for the node.

In Dalek, the nodes of a dataflow graph represent event templates: different occurrences of an event can be formed from its template. The leaves of the dataflow graph correspond to templates for primitive events (initial, independent sources of tokens) while the interior nodes represent templates for higher-level (dependent) events. A token corresponds to an *occurrence* of an event. It is thus a composite object: The information required to characterize this specific instance of the event, and to distinguish it from other occurrences of the same event, is carried within the token as its attributes.

When defining an event, an essential ingredient is the writing of its command block — Dalek language code that will reside at that event template's node in the dataflow graph. The primary purpose of this code, when it is interpreted after its node has been triggered, is to determine whether a valid instance of that event should be recognized. Successful recognition would allow tokens representing this particular instance to propagate further in the dataflow graph.

Since each instance of an event has its own attributes (located in Dalek's address space), and since each invocation of an event template's command block can have local variables as well, the commands for each event can be written in isolation from those for other events. These properties allow events to mirror the logical structures of an inferior program quite naturally. They also permit 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.

3.2. Primitive Events

In order for an instance of a primitive event to occur, it must first be explicitly raised by name. Initially, we designate primitive events as belonging to one of two classes, "bound" or "free-floating", depending on how they are raised. Later, in Section 6.6.2, we will introduce a third class of primitive events, the "loosely-bound" events. These are raised in a manner that is, in some sense, intermediate to the modes in which bound and free-floating events are raised.

3.2.1. Bound Primitive Events

Most commonly, the user specifies an **event raise** command in the command block of a breakpoint inserted in the inferior process. In such a case, we refer to that primitive event as being "bound" to that particular location in the code. Figure 3-1 illustrates one such example.

If desired, a given primitive event may be thus raised from several different breakpoints in the inferior program's source code. In this case, the user may wish to declare a separate attribute for the express purpose of distinguishing between those occurrences of this event that have been raised from different locations in the inferior program's source code. In the example of Figure 3-1, this has been accomplished by assigning to one of 'foo's attributes the line number in the source code that is being executed when the event

event define ‘foo (‘x_val, ‘attribute_2, ‘prog_line)
  # Specify the command block for this event:
  event set-attribute ‘attribute_2 *arbitrary_expression*
end # The other attributes are set when ‘foo is raised — see below.

break foofile.c : 35 # Set a breakpoint at line 35.
commands # Specify a command block for the newest breakpoint.
  silent # Suppress default announcement of encountering a breakpoint.
  event raise ‘foo (*x*, , **$line**(0))
  # Assign current value of *x* to ‘foo’s first attribute; and invoke a predefined
  # Dalek function so current line number is assigned to ‘foo’s third attribute.
  # The second attribute’s value is not defined by this **event raise** command.
  cont
  # Continue executing the inferior, after activity in the dataflow graph ceases.
end

Fig. 3-1 Dalek code to define and raise a *bound* primitive event.

is raised. The predefined debugger function, **$line**(N), returns as its value, the line number being executed by the calling procedure N frames up in the inferior’s runtime stack. Thus **$line**(0) returns the number of the currently executing line of source code. In a more complicated situation, it might be helpful to define another attribute, ‘filename, and assign it the value returned by a call to **$file**(0), a predefined debugger function analogous to **$line**().

3.2.2. Free-Floating Primitive Events

Less commonly, a primitive event may be free-floating — not associated with any fixed location(s) in the inferior program. For example, it can be useful to raise a certain primitive event only on those occasions when the inferior process receives a particular signal, regardless of which source code line happens to be executing when the signal arrives. Another method for coupling a free-floating primitive event to the inferior's behavior is for it to be raised from inside a freestanding **while—step** loop (i.e., a loop to single-step the inferior and *watch* its state, entered directly from Dalek's interactive command level). With this method, events can be raised whenever the inferior's state satisfies an *arbitrary* condition, rather than being limited to those conditions containing an implicit clause that narrowly restricts the value of the inferior's program counter register (as is essentially the case with events that are raised at breakpoints).

3.2.2.1. Free-Floating Primitive Events Raised in while—step Loops

We present the example of Figure 3-2 to illustrate a free-floating primitive event that is hooked to the inferior by means of a **while—step** loop. We define an event 'new_max_x whose purpose is to record a series of progressively greater maxima reached by the inferior variable *x*. The Dalek code in Figure 3-2 will make a permanent record in the event history of the maximum excursions taken by the inferior variable *x*, correlated with a measure (in terms of the number of source lines executed) of when such excursions occurred. Astute readers who perceive a potential flaw in the coding style of Figure 3-2 may rest assured that their concern will be addressed in Section 5.4.1.1.

Note that a minor variant of the code in Figure 3-2 is easily capable of determining the maximum depth ever reached by the inferior's runtime stack during the course of its execution. All that is required is a change of names — for example, substitution of "$sp" (denoting the stack pointer register on 680x0 machines) for "*x*", and replacement of the

```
# First initialize two global convenience variables:
set  $num_steps = 0   # The number of source lines executed.
set  $old_max_x = x   # The maximum value of x seen so far.

# Next define a primitive event, including names for its two attributes:
event define  ‘new_max_x  ( ‘new_x_val, ‘num_steps )
   # First determine whether we should recognize the present
   # situation as a valid occurrence of event  ‘new_max_x:
   # Dalek’s predefined function $get_attribute() returns the value of
   # an attribute bound to the current instance of this event.
   if  ( $old_max_x  >= $get_attribute (‘new_x_val) )
      event dont-propagate     # Not a valid instance of this event.
   else
      set  $old_max_x = $get_attribute (‘new_x_val)
                            # "Ratchet" up $old_max_x.
      printf  "Variable ‘x’ reaching a new maximum of %d\n", \
                           $get_attribute (‘new_x_val)
      # Note attribute ‘new_x_val will already have been instantiated by
      # the command (in the while loop below) that raises this event.
      # Although we could directly access the inferior variable x here,
      # that would be inefficient, requiring another UNIX context switch.
   endif
end    # End of this event’s command block.

# Now enter a freestanding watch loop to single-step the inferior program:
while ( $running )
   # Activate the event’s command block each time (also inefficient)!
   # Its first attribute is assigned the current value of x:
   event raise  ‘new_max_x  ( x, $num_steps )
   step    # Single-step the inferior process after ‘new_max_x finishes.
   set  $num_steps ++
endwhile
```

Fig. 3-2 Dalek code to illustrate a free-floating primitive event.

step command by **stepi**, to shorten the strides to the granularity of single machine instructions.[2]

3.2.2.2. Free-Floating Primitive Events Keyed to Signals

Another way that a free-floating primitive event can be linked to the inferior's behavior is shown in Figure 3-3. Gdb's original syntax included a command to

handle SIGNUM [stop/nostop] [print/noprint] [pass/nopass]

This command instructed gdb how to behave when it intercepted a signal bound for the inferior — whether to stop (i.e., resume interactive mode), print (i.e., notify the user), and/or pass the signal through to the inferior process. Dalek extends gdb's previous functionality by allowing a block of commands to be interpreted whenever the appropriate signal is intercepted. Figure 3-3 exploits this feature to record, via an event, each reception of some arbitrary signal.

Of course, if the user's application program had specified its own signal handling

```
handle $SIGNUM nostop noprint pass commands
   event raise 'sig_received ( $SIGNUM )
end   # End of this signal's command block.
```

Fig. 3-3 Another way to hook a free-floating primitive event to an inferior.

[2] If the Dalek code took **step**-sized strides, it would fail to notice changes in the inferior's stack depth when the inferior invoked functions residing in files that were compiled without the "-g" flag. Such functions would include all library routines and any procedures written in assembler.

procedure for $SIGNUM, then we might be able to achieve the same objective by raising the event from a breakpoint set within that procedure. However, some versions of UNIX decree that the user's signal-handler must reinstall itself every time it receives a signal if it wants future occurrences of $SIGNUM routed its way. On such systems, it would be safer to raise the event as shown in Figure 3-3, in case the signal-handler fails to reinstall itself, either by design or due to some bug. The method of Figure 3-3 is the *only* way to capture information regarding signals for which the inferior has *not* declared its own signal handling procedure (i.e., those signals that are routed to the default handler provided by the system, or those that are ignored altogether by the inferior).

3.2.3. Primitive Events: Attributes and the Flow of Tokens

We have demonstrated several ways in which primitive events can be coupled to the inferior so that they faithfully monitor its behavior. Regardless of the exact method, suffice it to say that somehow, activity within the inferior process can cause a primitive event to be raised, whereupon Dalek will begin to interpret the code in its command block. During this time, the inferior process remains suspended.

Dalek tacitly assumes that a successful interpretation of an event's command block is tantamount to recognition that a valid instance of that event has indeed occurred. To inform it to the contrary, once an event has been raised and Dalek has begun to interpret its command block, the occurrence can be repudiated by the **event dont-propagate** command.

If the code at a primitive event's node decides that a valid instance of the event has occurred, then (among its possible actions) it should ensure that appropriate values are assigned to the attributes that will be bound to this particular occurrence of the event. In the example of Figure 3-1, the first and third attributes of ‘foo are implicitly assigned values within the **event raise** command. Presumably it is more complicated to compute a value for the second attribute, so the command block for event ‘foo should contain code to accomplish that assignment.

If a primitive event template successfully achieves recognition, then once Dalek finishes interpreting its command block, parameterized tokens embodying that particular instance will be propagated to all higher-level events that depend on that primitive event — i.e., copies of its distinctive token will flow along dedicated arcs to all higher-level events connected to that primitive event in the dataflow graph. The arrival of one of these tokens at a higher-level node can trigger that event template, informing Dalek that the code for that higher-level event should be interpreted after that of any other, previously triggered events.

3.3. High-Level Events

During the ensuing discussion of high-level events, the reader may find it helpful to refer to Figure 3-4, a stylized dataflow graph representing an upcoming example.

Recall that in Dalek, the command block (i.e., the Dalek code) associated with an event template resides at a node in the dataflow graph. Incoming arcs to an interior (high-level) node 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 lower-level events’ tokens, after attaching to the high-level event’s incoming queues, can cause Dalek to activate that

Fig. 3-4 Memory allocation events embedded in a stylized dataflow graph.

event's command block so it can attempt to recognize an instance of the high-level event. Syntactically, in the definition of the high-level event, a minus sign prefixing the name of some lower-level constituent indicates that the arrival of a token from that constituent should *not* trigger the high-level event. The token will, however, be enqueued so that its attributes are available to a future activation of the high-level event.

When a high-level event is activated, 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 tokens currently present on its queues (or some subset thereof) comprise a valid instance of the high-level event. If such is the case, the code should instantiate the attributes characterizing this particular occurrence of the high-level event, either by copying the values of selected constituents' tokens' attributes, or by computing some function based on the available data. Other possible actions include printing a message, reestablishing interactive control while the inferior process remains suspended, and even modifying data or control flow in the inferior process. The code for the high-level event will then generally remove the selected constituents' tokens from its incoming queues so as to avoid the possibility of repeatedly recognizing the same set of tokens when it is subsequently activated again. When Dalek finishes interpreting the high-level event's code, tokens embodying this new event instance will propagate to any other high-level events that may, in turn, depend on it, and the recognition phase begins anew.

On the other hand, the high-level event's code might decide that an appropriate combination of constituents' tokens is not present on its incoming queues, or that their attributes fail to satisfy the desired relationships. Just as for a primitive event, the **event dont-propagate** command will suppress the default generation of tokens representing an occurrence of this high-level event, and the subsequent passage of those tokens to any higher-level events that depend on this one.

If an event occurrence is successfully recognized, and if its tokens trigger multiple higher-level events, then the order in which those higher-level events' command blocks are interpreted is determined only by the implementation (in this case, Dalek). When every triggered node has had its command block interpreted — i.e., the dataflow graph becomes quiescent — Dalek resumes its previous interpretation activity, beginning with the command following the one that raised the initial primitive event.

Normally, Dalek will sequentially interpret the entire command block associated with an event template before it begins to interpret the command block for any other event. However, exceptions to this general rule can occur. For example, there is nothing (other than good judgement and plain common sense) to prevent the user from including a **cont** command in the middle of an event's code. Should Dalek interpret such a command, it will immediately suspend the current thread of activity in the dataflow graph in order to resume execution of the inferior, which might hit a breakpoint. The commands associated with that breakpoint might raise another event, whose command block Dalek would then proceed to interpret. In such situations, Dalek's only guarantee to the user is that *eventually* (unless he issues a **set exit** command) it will resume all these partially-interpreted command blocks.

Thus, although the user *can* make any dataflow graph in Dalek *re-entrant*, the resulting semantics constitute a research topic, so this is *not* a dependable technique whose use we recommend. Because such matters are not germane to the present discussion, and because they comprise only one genus of a much broader class, treatment of this topic is deferred until Chapters 4 and 6.

3.3.1. Example — High-Level Event to Monitor Memory Allocation

To illustrate one extremely practical use of 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.

Figure 3-5 shows the definition of two primitive events, ‘malloc and ‘free, whose instances 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. **$quiet_finish**() is a predefined debugger function that allows the inferior process to resume executing until the currently selected procedure returns. At that point **$quiet_finish**() will also return, delivering that inferior function’s return value as its own. The "currently selected procedure" defaults to the currently executing procedure unless the user has explicitly selected a procedure higher up in the inferior’s runtime stack. As with most things in Dalek, such "selection" can be achieved programmatically as well as

---

```
#	By default, event ‘malloc is recognized each time it is raised;
#	the same applies to event ‘free.
event define ‘malloc (‘addr)
   event set-attribute ‘addr $quiet_finish()
end

event define ‘free (‘addr)
   event set-attribute ‘addr   * (char **) $arg(1)
end
```

Fig. 3-5 Two primitive events to help monitor memory allocation.

---

interactively.

Since *free* is a system-dependent library routine, it was compiled without benefit of the "-g" command line argument. Thus there is no symbolic name that designates its argument, and we must construct the user-defined debugger function **$arg**(N) to return the address of the N th argument to the currently selected inferior procedure. In this case, **$arg**(1) will return the address of the block being freed (i.e., *free*'s argument). Since different machine architectures realize runtime stacks differently, the details of **$arg**() are also system-dependent.[3]

Error detection is carried out in the high-level event ‘match, which keeps track of the addresses of all memory blocks that have been allocated but not yet matched with a deallocate request. Whenever a ‘free occurs, ‘match checks to see if a corresponding instance of ‘malloc inhabits its queue. If an instance of ‘malloc carrying the proper 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 in Figure 3-6, ‘match is triggered only by the arrival of a token from ‘free. The matching of ‘malloc and ‘free events is performed using a simple **while** loop that compares the attribute ‘addr carried by the new token from ‘free with those stored in ‘match's queue for previous instances of ‘malloc.

---

[3] Too often, authors evade difficult questions by stating only that the answers are system-dependent, without offering a general solution, much less any particular answer. Regarding the current issue, for a Sun3 running Sun UNIX 4.2 Release 3.4, one solution is:

```
udf  $arg ($num)
   func-return   $fp +  ($num + 1) * sizeof(int)
end
```

where "$fp" refers to the frame-pointer register on Motorola 680x0 architectures.

```
#       The high-level event ‘match depends on lower level events
#       ‘malloc and ‘free, but it is triggered only by tokens from ‘free.
event define ‘match (‘match_addr)  −‘malloc  ‘free
   {
   local $qindex, $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)
         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 dont-propagate
   else
      event set-attribute  ‘match_addr  $free_addr
      event remove ‘malloc, $qindex
      # Remove the $qindex’th token from ‘match’s queue of ‘malloc instances.
   endif
   # In both cases, remove first token from ‘match’s queue of ‘free instances.
   event remove ‘free, 1
   }
end
```

Fig. 3-6 Dalek code defining the high-level event ‘match.

The reader may notice an unfamiliar command in the Dalek code of Figure 3-6. To force an immediate exit from a **while** loop, we chose the syntax **wbreak** because **break** itself was previously defined to mean "set a breakpoint". Similarly, since the normal C syntax, "continue", is too similar to **cont**, and **next** in gdb’s original syntax means to single-step the inferior (treating function calls as one step) Dalek uses **wnext** to force a jump to the next iteration of a **while** loop.

3.4. A Database of Event Occurrences

Allowing the user to visually browse an event history database at any time has proven to be quite helpful. The unabridged syntax to accomplish this is:

event history [h / p]

The optional "h" ("p") parameter invokes a filter through which only higher-level (primitive) event occurrences can pass en route to the display. The user can also narrow the focus even further by selecting certain events by name. Instances of the qualifying events, including the values of all attributes, are presented in temporal order beginning with the most recent occurrence.

Experience has shown that this tool to filter and summarize relevant events in the inferior's execution can promote clearer thinking on the part of a user regarding exactly *what* occurred, and *when*. Freed of the burden of such bookkeeping details, the user can employ his faculties more productively — in coming up with new hypotheses to explain the perceived behavior, or, given an existing debugging hypothesis, in developing methods to acquire sufficient data, in a subsequent program run, to either confirm or disprove that hypothesis.

We decided it might be even more useful to support *programmatic* modes for accessing the values of attributes long after an event had occurred. To integrate such programmatic access to historical events with Dalek's other features, the event history database was made to appear conceptually and syntactically similar to an ultra high-level event. For example, it seems to have named queues containing tokens representing occurrences of different events. The length of a named queue corresponds to the number of times the event with that name has occurred. The length of a queue in the event history can be determined by invoking the predefined debugger function **$hlen**(), instead of by invoking **$qlen**() as one would do within the command block of a genuine high-level event. Similarly, whereas a genuine high-level event's command block would invoke

$time (‘constituent_event_name, queue_position)

to return the timestamp of a particular instance of a constituent in one of its queues, we can invoke

$htime (‘event_name, queue_position)

at any point, even from the interactive command level, to garner similar information from the event history database. Of course, since **$htime**() and **$time**() will access physically different queues (even when given the same ‘event_name), their queue_position arguments will seldom correspond since tokens *cannot* be removed from the event history,[4] whereas tokens from lower-level constituents that have been successfully recognized as comprising a high-level event instance typically *are* removed by that high-level event from *its* queues.

Thus among its predefined debugger functions, Dalek incorporates several (e.g., **$hlen**(), **$htime**(), and **$hvalue**()) allowing the user to programmatically access all recorded event information. Although our current syntax for accomplishing this functionality is

[4] Although when the inferior program is re-started, this typically clears the entire event history database unless the user has explicitly specified that all occurrences of certain events should be preserved.

fairly primitive, it has proved adequate to demonstrate the utility of programmatic modes for accessing the event history.

For instance, the memory allocation example of Figure 3-6 could be extended to determine whether the reason for a mismatched *free()* is that a particular dynamically allocated block is being freed a second time. This enhancement would require only a slight modification to event ‘match: A new loop should be added so that whenever an unmatched ‘free is found, the loop simply searches the event history database for the most recent occurrence of a ‘match whose ‘match_addr attribute is equal to $free_addr. With additional attributes in the primitive events ‘malloc and ‘free to record the names of the inferior procedures that invoke *malloc()* and *free()*, we could make the crucial transition from merely detecting the *symptoms* of a bug to discovering the location(s) in the source code of its *cause(s)*.

Searching the event history database is useful in many different debugging contexts. Suppose, for example, the user detects that the contents of an allocated structure have been corrupted. At that point, he can type in a **while** loop to search the event history list to determine whether the address of that structure was ever freed. If so, knowing the identity of the module that allocated that address before it was freed could help to pinpoint any dangling references that may have caused the data corruption.

3.4.1. Subsequently Checking Assertions via the Event Database

As an example of what can be accomplished "after the fact" by intelligent programmatic access of the event history, suppose that the inferior has been running for quite some time, and that it has exhibited some anomalous behavior. Now at this point, perhaps the user begins to suspect that the errant behavior may have been caused by a particular data structure in the inferior containing duplicate items at some point during its life. But suppose that so far, event attributes have *only* been used to record the key of each item inserted into or deleted from that structure. Thus the condition of uniqueness among that data structure's elements has never been explicitly checked by the debugger.

Obviously one way to confirm or deny the uniqueness hypothesis is to write a user-defined debugger function that will check for duplicates in that inferior data structure, and to call this function every time the inferior inserts an item into that structure. But this method necessitates re-starting the inferior process from the very beginning. While this might be feasible in many cases, it could also be difficult or time-consuming in other situations. In addition, some sequential programs are not guaranteed to behave identically during two different runs — perhaps they handle asynchronous interrupts or use the system clock to seed a random number generator.

The user can avoid rerunning the inferior process by *deriving* the desired information directly from the "raw" data stored in the event history: A pair of nested **while** loops can be used to iterate over the event history to discover whether that inferior data structure has ever contained duplicate items. For each inserted item, we merely need to confirm that any subsequent re-insertion of an item carrying that same key was preceded by a corresponding deletion, and that the timestamp of this deletion lay between those of the two insertions. This situation gives us an excuse to write **$deleted_during_interval**(), a user-defined debugger function shown in Figure 3-7. It calls some of Dalek's predefined debugger functions that access the event history database. In a manner exemplified by

```
# Assume two events with the following names were defined initially:
event define  'insert  ( 'key )
   # An empty command block will suffice.
end
event define  'delete  ( 'key )
   # An empty command block will suffice.
end

# We next need to specify a user defined function:
udf  $deleted_during_interval ( $current_key, $time1, $time2 )
   # An item with $current_key was inserted at $time1 and again at $time2.
   set  $del_queue_index = 0
   while  ( ++ $del_queue_index <= $hlen ('delete) )
      if ( $htime ('delete, $del_queue_index)  <  $time1 )
         # This occurrence is too early to be relevant.
         wnext # We will advance to the next loop iteration.
      elseif ( $htime ('delete, $del_queue_index)  >  $time2 )
         # This and all later occurrences are beyond the relevant interval.
         func-return 0 # Report $current_key was not deleted in this interval.
      endif
      if ( $current_key == $hvalue ('delete, $del_queue_index, 'key) )
         # An item with that key was deleted during the interval.
         func-return  1
      endif
   endwhile
end
```

Fig. 3-7 A user-defined function to help search the event history list.

Figures 3-7 and 3-8, Dalek can assist the user in supporting or discarding newly formed hypotheses on the spot, rather than always forcing the user to redefine events with extra attributes (or additional restrictions on their relationships) and then to rerun the inferior program. Notice that the code of Figures 3-7 and 3-8 is completely generic. That is, it will function properly regardless of whether the data structure of interest in the inferior is implemented as a linked-list, binary tree, or hash table.

Once again, if the user had the foresight to capture the filename and line number of the *calling* function as attributes of primitive events ‘insert and ‘delete, this information would significantly help to localize the *cause* of the bug whose symptoms manifested as duplicate keys appearing in the inferior’s data structure.