CHAPTER 6

Semantic Integrity, and Code Interference from Colliding Coroutines

So far, the reader should have acquired the impression that all of Dalek's new features are well integrated; that is, they work well together and provide a friendly, supportive debugging environment with no major "gotchas" lurking to ensnare the unwary user. Several semantic issues were raised in Chapter 5, and were (we think) adequately addressed there as well.

Previous examples in Chapter 3 have shown how a primitive event could be either "free-floating" or "bound" to specific location(s) in the inferior's code. A bound event is (conditionally) raised only at certain breakpoint(s), whereas a free-floating event may be raised whenever the inferior's state satisfies some more general condition that may be independent of the current value in its program counter register. One example in particular, Figure 3-2, illustrated how such a free-floating primitive event could be (relatively *in*efficiently) coupled to the inferior's state by wrapping the command that raises it (**event raise**) inside a **while—step** loop. Such a loop would propel the inferior's execution and might also monitor the status of appropriate conditions for the event to be recognized. We describe a coupling between the inferior process and a free-floating primitive event as "relatively efficient" when the event is seldom raised unnecessarily. In such a case, at least some of the code for recognizing valid instances of that event would appear inline, within the body of the **while—step** loop.

Other examples (e.g., Figure 4-5) have shown how Dalek code (perhaps in an event's command block or associated with a breakpoint) could **call** an inferior function in order to check a Boolean condition or to provide a value for an attribute.

We must now admit that such examples may have been somewhat misleading. This is not to imply that the previous examples do not work as shown. The previous examples do indeed work in isolation, but combining some of them can reveal the presence of a fundamental conflict that is inherent in an oversimplified view of the underlying semantics. We realized the magnitude of this problem only after gaining considerable experience as users of Dalek, although in retrospect, it could (and perhaps *should*) have been foreseen earlier in Dalek's design stages.

6.1. A Framework for Analyzing the Formal Semantics of a Debugging Activity

Although debugging, being a practical activity, must concern itself with certain *pragmatic* goals — and with achieving those goals efficiently — our paramount concern at this stage is not with ultimate goals, nor with raw speed, but rather with *accuracy*. In order to define that term, we shall limit our present discussion to just those debugging cases where the user does *not* intend to change the "natural" course of the inferior's execution. Of course, for "natural" to be well-defined, we must also restrict our attention to only those inferior processes whose behavior, given the same input data, is *repeatable*. Thus they must be deterministic and insensitive to the passage of real time, as well as isolated from any asynchronous interrupts in their external environment. We shall employ two different, but closely related measures of accuracy, both of which will assist us in getting a handle on the *formal semantics* of the debugging activity. First, since we have temporarily limited ourselves to those debugging activities where the user does not *intend* to change the inferior's natural course, then it certainly makes sense to say that if some aspect of the debugging activity *should* happen to change that natural course, then that perturbation would be *unintentional*.

Thus our first measure of accuracy: The inferior must not deviate from its natural (repeatable) course of execution. Having ruled out all other external influences, if such a deviation should occur, then it could only have been caused by the debugging activity. Because of the obvious parallels, we will term any such perturbations first-order Heisenberg effects [Heis27] [Heis58]. That is, in attempting to observe a phenomenon, we may inadvertently disturb it.

Now "accuracy" itself implies some sort of measurement; it hints at the degree to which a *recorded* observation *corresponds* to some actual property of the phenomenon under investigation.

In our case, assuming that our actions in observing a phenomenon do not alter its actual behavior, this correspondence will be between certain items recorded by Dalek in the event history and the inferior's *actual logical timeline*. That actual logical timeline can formally be represented as a sequence of memory images — snapshots of the inferior's entire address space and the contents of its CPU's registers at each logical instant.[1] In practice, we could realize that actual logical timeline by ordering a core dump after the inferior executes each machine instruction. One purpose of the event history is obviously to "compress" this data physically. But another, completely orthogonal purpose is to "filter" it semantically. In contrast to physical filtration, our method not only filters cer-

---

[1] Such snapshots should technically include images of all *files* currently opened by the inferior.

tain items *out*, but also *inserts* new items into the event stream.

Therefore, if we wish to speak precisely of the notion of a "correspondence", we must ignore instances in our history of any events whose "meanings are semantically open to interpretation". In order to clarify this, we employ the language of logical positivism [Ayer36], for which we ask the reader's forbearance. Now because we permit the user to define and form his own abstractions via the mechanism of events, it will not, in general, be true that every event occurrence will correspond to some raw, *observable* property of the inferior that can be *localized* at a single point on its logical timeline, and at a single memory location.

Clearly, if all attributes of an event are intended to correspond to the values of various particular memory locations at the same well-defined instant, then we can easily specify an "effective procedure" by means of which it will be possible for several *different* observers to map an occurrence of that event in the event history to the same observable characteristics of the inferior's actual timeline. For such cases, it will thus be possible objectively to *verify* the accuracy of the correspondence between the inferior's behavior and these recorded measurements. Those events whose *semantics* are indeed localized to such a degree we shall include among the "objective facts" regarding the inferior's behavior. About such matters, there can be no disagreement between any two observers, e.g., regarding the fact that the value of *x* was 80 at logical time t_{113}.

Certainly there may be other, less localized events that qualify as objective facts, for example, the fact that *foo*() was invoked and then, at a subsequent point on the timeline, returned. What we wish to avoid is the possibility of two different observers, having defined the same primitive events, recording the behavior of the same inferior computation and yet *disagreeing* about what actually occurred.

For example, one observer may define a high-level event he calls "thrashing", and claim that its onset was observed at logical time t_{101} in the inferior. If we admit "thrashing" as an objective fact, and admit that his event history is accurate, then *if* another observer has defined a similar high-level event, we force both observers to employ the same semantic interpretation function, i.e., the same meaning, for the event, ‘thrashing.

Not being born-again logical positivists, we strongly encourage the user to define *any* event abstractions whatsoever — including purely *subjective* interpretation functions — if they assist him in understanding the inferior’s behavior. After all, since debugging is a practical activity, our motto must be, "Whatever works". Nevertheless, we shall banish all such vague, subjective entities to the realm of pragmatics, since we must exclude them from our formal semantic analysis.

Finally then, this must be our second measure of accuracy: How closely do the "objective facts" among our *recorded* events correspond to unambiguous, observable characteristics of the inferior’s actual logical timeline?

Although we believe it is possible to state an effective procedure that will allow one to discriminate between "objective" and "subjective" facts regarding an inferior process during the debugging activity, we have no desire to attempt to do so now. To do that would probably require the introduction of some new formal notation, whose syntax (as in most formal semantic analyses) would obscure the underlying, intuitive semantics of the situation. Instead, we are content to wave our hands and say only that it is the more primitive instances among the recognized event occurrences comprising the event history that are the more direct "measurements" of the inferior's behavior. While they may admit of various interpretations, they, among all the events, are the most "widely trusted". That is, they are the least likely to have been semantically compromised by some particular user's possibly ambiguous, ill-defined, or subjective semantic interpretation functions. Looked at another way, the primitive event instances comprise the (incrementally) "given facts" in a knowledge base, and the higher-level event occurrences are (incrementally) "derived facts", added to the knowledge base by applications of the "production rules", i.e., the high-level events' recognition code. Thus, to oversimplify matters, we might say that because primitive events are somehow closer to the inferior's actual history, we might equate primitive events with objective facts and equate higher-level events with progressively more complex (and therefore error-prone) or subjective interpretations of those facts.

Now we must assume that the user's event recognition code for the *primitive* events, at least, is itself free of any basic, logical bugs. Why, then, need we even concern ourselves with this second measure of accuracy? If the objective facts recorded in the event history should ever fail to correspond exactly with those aspects of the inferior's actual behavior they were *intended* to measure, then can we not blame any and all such discrepancies on the user? Unfortunately, we believe that question must be answered in the negative. Dalek's language is expressive enough that we believe certain unforeseen and unintended interactions between language features — henceforth called "code interference" — *cannot* be blamed on the user. Therefore, we are extremely concerned with this second measure of accuracy. In fact, that is, in essence, the subject of the present chapter.

Recall that, when our attempts at observation disturbed the inferior directly, we termed those "first-order" Heisenberg effects. Now in certain circumstances, it is not only possible, but probable, that in attempting to measure one aspect of the inferior's behavior, we may disturb the results of *another observation*, even though the inferior's *actual* behavior is not perturbed in the slightest. We will term such disturbances higher-order or "meta-Heisenberg" effects.

Finally, let us introduce one further term into our discussion: We shall say that whenever *both* criteria of accuracy have been met — whenever neither first-order nor higher-order Heisenberg effects have occurred — then the event history demonstrates that property we call *semantic integrity*. That is, the objective aspects of the event history accurately reflect *exactly* that behavior in the inferior that the user *intended* to capture. Is such a thing possible? Well, the trivial case is one in which the user has defined no events whatsoever. In that case, the (empty) event history does indeed enjoy the property we call semantic integrity. Given that terminological background, we can now proceed to less trivial cases.

We will classify problems arising from code interference into two categories based on the nature of their symptoms. The heading of *false negatives* includes situations, such as those described in the next section, in which Dalek might neglect to record an occurrence of some free-floating primitive event because the inferior is given free rein to proceed — despite the fact that doing so inadvertently decouples that event's "hooks" from the inferior's behavior. *False positives*, discussed in Section 6.2.2, refer to event occurrences that are somehow recognized by mistake.

```
# First define two simple, primitive events.
event define  ‘x_modified  ( ‘new_value )
   # Recognized by default every time it is raised.
end    #  This event will be free-floating.

event define  ‘foo_called  ( ‘formal_parm )
   # An empty command block will suffice here too.
end    # This event will be bound to a specific inferior location.

# Specify a breakpoint so the latter event is raised whenever foo is called.
break  foo
commands
   silent
   event raise  ‘foo_called ( foos_parm )
   cont     # Allow the inferior to resume.
end

# Initialize a global convenience variable.
set $old_x_value = x

# The user now enters a  "freestanding" loop to single-step the inferior:
while ( $running )
   step           # Single-step the inferior process.
   if  ( $old_x_value != x )
      event raise  ‘x_modified ( x )
      set $old_x_value = x    # Update global convenience variable.
   endif
endwhile
```

Fig. 6-1 Dalek code exemplifying conflicting semantic assumptions.

6.2. Command Block Interference Leading to False Negatives

The problem of false negatives is exemplified by the very simple debugging code of Figure 6-1. The crux of the matter, as shown by the code near the bottom of that figure, is that the user intends to **step** the inferior process, watching for a given condition to be satisfied. If and when that condition becomes true, he wants to raise a free-floating primitive event to record that fact, then proceed within the **while—step** loop, watching for the next occurrence of that event. To simplify the exposition, we are ignoring, for the moment, the concerns discussed in Section 5.4.1.1 regarding the possibility that the referent of the name "*x*" may change as the inferior crosses different scoping contours during the course of its execution.

At some point, the user will **step** on his own breakpoint (sic!), so Dalek will stack the pending commands in the **while—step** loop and begin interpreting the breakpoint commands. The final command at the breakpoint instructs Dalek to **cont** the inferior. What will that do to our vigilant monitoring by the conditional statement embedded in the **while—step** loop? If no other breakpoint is subsequently encountered, or if the command block of every subsequent breakpoint also ends with a **cont** command, Dalek will not be able to resume its interpretation of the **while—step** loop until *after* the inferior process terminates! Clearly such was not the user's original intention when he wrote the debugging code in Figure 6-1. Dispensing with the **cont** command altogether might seem to solve the problem in this particular case. However, if the user later decided he did not want to use the **while—step** loop, then when the inferior encountered the breakpoint, it would remain suspended after event 'foo_called was raised. Execution of the inferior would not resume until the user *interactively* issued a **cont** command. Thus this "solution" would effectively force the user to rewrite all breakpoint command blocks whenever he changed his mind about employing a **while—step** loop. Such a property cannot be considered even remotely user-friendly.

6.3. Command Blocks May Execute as Re-Entrant Coroutines

Are **while—step** loops then fundamentally incompatible with breakpoint commands? Must we choose between bound and free-floating primitive events? Has much of our work been for naught? Fortunately, the answer to all these questions is an emphatic, "No".

By ignoring an important complicating factor in previous chapters, we may have been guilty of encouraging a false simplicity in the reader's view of the coherence of certain debugging constructs. The reason for our de facto oversimplification was that we wanted to demonstrate how *easily* the user can write debugging code. Yet unbeknownst to the user, such debugging code might operate in practice as a *set* of *re-entrant coroutines*! Viewed from the perspective of the operating system, there are really only two coroutines, Dalek and its inferior. Yet from the user's point of view, *every command block* that might cause the inferior to resume execution, or that might be activated as a result of the inferior hitting a breakpoint, should be viewed as a potential coroutine, in addition to the inferior process itself.

For example, the debugger process might delegate control to the inferior process via a call to **$quiet_finish**() part way through command block *A*, and might "prematurely" regain control because the inferior encounters a breakpoint before reaching its assigned goal. If command block *B* is associated with that breakpoint, this would cause the debugger to suspend its interpretation of *A* in order to begin interpreting block *B*. Next, a command in *B* might **step** the inferior onto another breakpoint, leading in turn to *B*'s suspension and causing control to re-emerge in command block *C*. Command block *C* might then **call** an inferior function, but again the inferior might encounter a breakpoint before that **call** can complete. This latest breakpoint may be one member of a broadcast set associated with command block *A*, causing Dalek to suspend block *C* and begin a *re-entrant* interpretation of block *A*, where this saga began.

Certainly by inspecting the text of, e.g., command block *A*, it would not be readily apparent under what circumstances it would transfer control to another command block, much less to block *B* in particular. Nor would it be obvious that a re-entrant interpretation of block *A* may be initiated before control resumes at the line after the call to **$quiet_finish**() in the original activation of *A*. For that matter, control may *never* resume in *A*'s original activation, if a dynamically-succeeding control block contains a command to, e.g., "**set exit** 999".

Such transfers of control — which permit the focus of Dalek's interpretation to "tunnel" through the inferior and re-emerge at an arbitrary breakpoint command block — are disquietingly reminiscent of wormholes through hyperspace, and stand in sharp contrast to more tractable control transfer mechanisms such as the calling of functions and the propagation of tokens through a dataflow graph. Thus if the user did not thoroughly understand every aspect of the inferior program's dynamic behavior *before* he began writing his debugger program, it is unlikely he will be able to reliably predict the inferior's effect on the behavior of any supposedly independent command blocks comprising his debugger program as they "execute" in their roles as conjugate coroutines.

Suppose we step back and adopt the perspective of the operating system, whereupon the complicated set of re-entrant coroutines we have just described coalesces into merely two coroutines, only one of which (the inferior) can be made re-entrant (by Dalek using the **call** command). Yet even then, the trajectory followed by a single thread of control as it wends between the two different processes can be quite complex and difficult to predict in advance, especially given the expressive power of Dalek's language.

The debugger language furnished by Dalek is, after all, an application-specific language. The requirement of "full programmability" ensures that a subset of its syntax is as powerful and expressive as typical general-purpose programming languages, yet the more basic, meat-and-potatoes requirement of controlling an inferior process necessitates certain specialized constructs not commonly found in general-purpose programming languages. It may be worthwhile at this point to reiterate something first mentioned in Section 5.3.1 on the structure and complexity of the debugger's implementation:

> . . . other commands, e.g., **cont**, **step**, or **call**, entail that execution of the inferior be resumed. The ability to carry out such commands helps to determine, and to complicate, the structure of gdb's code below its main, interpretive loop. Enhancements related to this ability have also complicated the code for Dalek.

Not only has the existence of these specialized commands profoundly impacted the design of Dalek's own source code, but, as we shall see in the next section, their influence extends even to the design of a user's debugging code that is to be interpreted by Dalek.

6.4. Steps Toward a Solution

Keeping in mind the tension between breakpoint command blocks and the **while—step** debugging construct (from which arises the discord between bound and free-floating primitive events), we will first show how the sophisticated user can harmonize these seemingly disparate constructs by following certain conventions when writing his debugging code. Since these coding conventions may be somewhat arduous, we will later proceed to suggest how some of the safeguards they afford might be incorporated into the debugging language itself, so that this functionality would automatically be made available, even to novice users who may have an oversimplified view of the problem.

6.4.1. The User Can Prevent Many Common False Negatives

Let us begin by attacking an easier problem than that exemplified by Figure 6-1. Suppose the user desires to track changes to the values of two different global variables. Clearly, the code in Figure 6-2 is *not* the best way to achieve this. There should be near-unanimous agreement that the code in Figure 6-2 is nothing more than a stupid blunder by the user. The first pass through the outer **while** loop will eventually execute **watch-y**, after which the outer loop will never again regain control so it can check whether *x* has been modified. Exactly why should the user be blamed for this? Why not blame Dalek's language instead? Although opinions may differ, we believe the most important reason is that **watch-y** appears statically, *by name* in the outer loop. Given this clue, it is then a very simple matter for the user to inspect the commands contained in **watch-y**, in case he has forgotten the definition he originally specified. Thus the situation in Figure 6-2 is akin to the user writing a pair of directly-nested **while** loops — any idiot ought to be able visually to *see* what the consequences will be.

In contrast, if we remove the **watch-y** command from the lower loop in Figure 6-2, and instead that loop's **step** command causes the inferior to execute a breakpoint whose com-

```
# A User-Defined Command is constructed:
udc  watch-y
   while ( $running )
      step    # Single-step the inferior process.
      if  ( $old_y_value != y )
         event raise  ‘y_modified ( y )
      endif   # (Let ‘y_modified decide if $old_y_value should be reset.)
   endwhile
end

set  $old_x_value = x
set  $old_y_value = y
# The user now enters a freestanding loop containing his new command:
while ( $running )
   step          # Single-step the inferior process.
   if  ( $old_x_value != x )
      event raise  ‘x_modified ( x )
   endif     # (Let ‘x_modified decide if $old_x_value should be reset.)
   watch-y      # Interpret the new User-Defined Command.
endwhile
```

Fig. 6-2 A *wrong* way to monitor two free-floating primitive events.

mand block then activates **watch-y**, most people would be reluctant to criticize the user for overlooking something "obvious. In fact, given our initial postulate that the user is *not* intimately familiar with every subtle nuance of the inferior’s dynamic behavior (otherwise why would he need a debugger), we might even be justified in describing **step** and its ilk as *context-sensitive* commands. In the best of cases (i.e., even if no breakpoints were inserted in the inferior), **step** would still have to be classified syntactically as a *meta-variable* ranging over the lines of source code in the inferior program: at one point it denotes, e.g., line 35 in the inferior program, at the next point it has come to denote line 36 or, e.g., line 451, if line 35 contained code to call another inferior procedure located there. Of course, once we insert breakpoints in the inferior and permit the user to include commands such as **step**, **next**, **cont**, or **finish**, in the midst of arbitrary command blocks, such commands are no longer merely meta-variables ranging over the *inferior's* code, but have now become *aliases* capable of ranging over the "names" of the breakpoint command blocks in Dalek as well!

Given the opacity of commands like **cont** and **step**, whose interpretation permits control to "tunnel" through the inferior and re-emerge at an arbitrary breakpoint command block in Dalek, the user would be wise to practice "defensive programming" — i.e., to assume the worst, namely that a **step** command could execute *any* breakpoint in the inferior. Thus *every* breakpoint should expect to be stepped on by a **while—step** loop at some point in its life, so the user should code every breakpoint's command block accordingly. As a first approximation, we reproduce the code of Figure 5-4 as Figure 6-3, but include one minor name change to clarify that this convention is intended to provide another benefit in addition to the avoidance of tail recursion. As shown in Figure 6-3, by merely maintaining the global Boolean convenience variable, $WHILE_STEP, the user can easily ensure that any newly-encountered breakpoint command blocks that Dalek interprets will not cause gaps in the vigilance of a "dynamically enclosing" **while—step** loop. Naturally when coding such loops, the user must remember to set $WHILE_STEP before entry to the loop, and to reset it upon exiting the loop.

The breakpoint coding convention of Figure 6-3 clearly resolves the most egregious forms of debugging code interference. By practicing this discipline, the user can prevent

```
# To avoid both tail recursion and blatant code interference:

commands
   . . .        # Main body of command block.
  if ( $WHILE_STEP )
    set exit 1    # Avoid the cont command below, thus allowing
  endif    # a stacked command block to resume stepping the inferior.
  cont            # Continue the inferior process.
end    # End of block of breakpoint commands
```

Fig. 6-3 Breakpoint coding convention to mesh with **while—step** loops.

most "false negative" misinterpretations of the event history; that is, he can ensure that with regard to event occurrences, absence of evidence *will indeed* constitute evidence of absence.

6.4.2. The User Can Prevent Most Common False Positives

But having thus dissipated the tension between bound and free-floating primitive events, we are not yet permitted to rest on our laurels. Unfortunately, "false positives" can still crop up in the event history. To understand how this might occur, we temporarily and slightly relax our restriction on altering the "natural course" of the inferior's execution, and allow the user to do so under strictly controlled conditions. We invite the reader to consider the debugging code shown in Figure 6-4.

Figure 6-4 shows two simple primitive events that are bound to specific locations in the inferior's source code. The code in event ‘graph_modified conditionally calls three dif-

```
# This event characterizes a graph whenever it gets modified:
event define  ‘graph_modified  ( ‘node, ‘distance_metric, ‘cyclic )
   event set-attribute ‘cyclic,  has_cycle (start_node)
   # We assume has_cycle() returns 0 if the graph is acyclic.
   if ( $get_attribute ( ‘cyclic ) )
      event set-attribute ‘distance,  graph_diameter (start_node)
   else
      event set-attribute ‘distance,  tree_height (start_node)
   endif
end

break modify_graph
commands
   event raise  ‘graph_modified ( start_node, , )
   # To avoid both tail recursion and blatant code interference:
   if  ( $WHILE_STEP )
      set exit 1    # Avoid the following cont command.
   endif
   cont             # Continue the inferior process.
end

event define  ‘has_cycle_invoked  ( ‘node, ‘caller )
   event set-attribute  ‘caller,  $func ( 1 )
end

break has_cycle
commands
   event raise  ‘has_cycle_invoked ( start_node, )
   # To avoid both tail recursion and blatant code interference:
   if  ( $WHILE_STEP )
      set exit 1    # Avoid the following cont command.
   endif
   cont             # Continue the inferior process.
end
```

Fig. 6-4 These two bound primitive events produce ‘false positives’.

ferent inferior functions in order to perform its mission. But when it calls *has_cycle* (), the inferior process will encounter a breakpoint located in that function and the corresponding breakpoint commands will raise an event to record the fact that *has_cycle* () was invoked. Yet this invocation is in a very strong sense "unnatural"; i.e., it is solely an artifact caused by Dalek interpreting the user's debugging code and would not occur in an unobserved run of the inferior program. Such Heisenberg effects are undesirable, to say the least. Yet if a user feels it necessary to thus alter the inferior's natural course, notice that he has made the inferior re-entrant under strictly controlled conditions. For purposes of *measurement*, he chose to initiate a new, temporary thread of control in the inferior. The user desired access to certain information, and employed pre-existing routines in the inferior to gain that information. The main thread of control in the inferior has not changed; it can be resumed and will continue along its natural course.[2] Thus our new, re-entrant thread is really just an offshoot from the inferior's natural course that terminates in a dead end. Is there some way that the user can employ inferior routines for just such limited purposes, without these spurious threads of control being reflected in the history?

Fortunately the problem of false positives admits a solution very similar to that for false negatives. Again the key is defensive programming. Figure 6-5 illustrates one coding convention that resolves the issue.

---

[2] Clearly, if such a re-entrant thread were to alter any global data, we would not consider that thread to be "strictly controlled". It is the user's responsibility to determine whether it is safe to **call** a particular inferior routine re-entrantly.

```
# To prevent "false positives", we need only add code guarding entry
# into an event’s command block.  No changes are required to the
# code in the breakpoint command blocks that raise these events.
event define  ‘graph_modified  ( ‘node, ‘distance_metric, ‘cyclic )
   # First determine whether this invocation is itself a bona fide one:
   if ( $artificial_call )
      event dont-propagate
      set exit 1     # Do not proceed with the normal command block.
   endif

   # Since this event calls inferior functions, we must set a global
   # convenience variable just in case other breakpoints are encountered.
   set  $artificial_call = 1          # Practice defensive programming!
   # Now we can proceed with the core of this event’s command block:
   event set-attribute ‘cyclic, has_cycle (start_node)
   # We assume has_cycle() returns 0 if the graph is acyclic.
   if ( $get_attribute ( ‘cyclic ) )
      event set-attribute ‘distance, graph_diameter (start_node)
   else
      event set-attribute ‘distance, tree_height (start_node)
   endif
   set  $artificial_call = 0          # Remember to reset this global!
end

event define  ‘has_cycle_invoked  ( ‘node, ‘caller )
   # First determine whether this invocation is a bona fide one:
   if ( $artificial_call )
      event dont-propagate
      set exit 1     # Do not proceed with the normal command block.
   endif

   # The normal command block is then placed after the above guard:
   event set-attribute  ‘caller,  $func ( 1 )
end
```

Fig. 6-5 Coding convention to prevent ‘false positives’ in the event history.

6.4.3. Limitations of the Two Coding Conventions

The two coding conventions we have introduced so far rely on a false dichotomy between the purposes of commands within events and those associated with breakpoints: A breakpoint's command block (typically as its final act) is responsible for propelling the inferior process on its way; in contrast, an event's command block typically takes no responsibility for propelling the inferior along its "natural" course. Under normal circumstances, the mission of an event's command block is not to change the inferior's state, but rather is to record observations that characterize the inferior's "natural" behavior.

Opportunities for code interference are not strictly limited to those shown so far. For example, the event history would still be subject to "false negative" misinterpretations if the command block for an event were to set one of its attributes to the value returned by a call to **$quiet_finish**(). If a command raising that event were dynamically inside a **while—step** loop, then the call to **$quiet_finish**() would cause the inferior to execute an arbitrary number of source code lines, during which time the **while—step** loop would be oblivious to any changes in the inferior's state. If a condition for which the loop was watching occurred during this period *and remained true*, the loop would eventually notice, once it was resumed, and would take some action. But that action might no longer be appropriate given that other parts of the inferior's state could also have changed considerably since that condition was first satisfied. Equally worrisome is the possibility that one of the relevant conditions could be *momentarily* satisfied, but might revert before Dalek resumed interpretation of the watching loop.

Still, the two coding conventions shown earlier would suffice to prevent the most common sources of code interference. If other potentially-interfering constructs were employed by the user, they could be protected by similar coding conventions. Although such special cases are too numerous to examine individually, it might be instructive to provide an interference-free solution to the problem of an event setting an attribute by calling **$quiet_finish**(). Figure 6-6 shows a definition of event ‘foo that could interfere with the mission of a dynamically-enclosing **while—step** loop. An alternative, shown in Figure 6-7, avoids the problem entirely by virtue of the fact that, since no longer does an event call **$quiet_finish**(), no longer does an event resume execution of the inferior process. Thus in Figure 6-7, the **while—step** loop is the *only* construct that propels the inferior along the course of its execution. For the second version of event ‘foo to achieve the same functionality as the first definition, ‘foo itself has been promoted to a high-level event. Two additional primitive event constituents provide their attributes to ‘foo. The

```
event define ‘foo ( ‘foos_initial_parm, ‘foos_return_value )
    event set-attribute ‘foos_initial_parm  parm
    event set-attribute ‘foos_return_value  $quiet_finish()
    # Calling $quiet_finish() could cause the watching loop to
    # miss an occurrence of some free-floating primitive events.
end

break  foo
    event raise ‘foo ( )
end
```

Fig. 6-6 This formulation of event ‘foo could cause interference.

```
# We can avoid interference by making ‘foo a high-level event,
# with two new bound primitive events as constituents, one per attribute:

# Define the high-level event to gather attributes from its two constituents:
event define ‘foo ( ‘foos_initial_parm, ‘foos_return_value )   \
            - ‘foo_starting,    + ‘foo_ending
   event set-attribute ‘foos_initial_parm  \
                        $value (‘foo_starting, 0, ‘foos_initial_parm)
   event set-attribute ‘foos_return_value  \
                        $value (‘foo_ending, 0, ‘foos_return_value)
   event remove ‘foo_starting, 0
   event remove ‘foo_ending, 0
end

# Now define the two primitive event constituents.  The first is simple:
event define ‘foo_starting ( ‘foos_initial_parm )
   event set-attribute ‘foos_initial_parm  parm
end

break  foo
   event raise ‘foo_starting ( )
end


# The definition of the second primitive event constituent is simple,
# but it may be bound to several different points in the inferior:
event define ‘foo_ending ( ‘foos_return_value )
   event set-attribute ‘foos_return_value  ret_val
end

# Broadcast a breakpoint to all "return" instructions in the codespace of foo():
asm-break  &foo,  $func_length ("foo"),  rts
   event raise ‘foo_ending ( )
end
```

Fig. 6-7 Avoiding interference through a new, high-level event.

crucial difference is that ‘foo_starting avoids calling **$quiet_finish**(). Instead it terminates, allowing the inferior to traverse the remainder of procedure *foo*() under the direct control of the watching loop. Thus we require a second breakpoint (set), broadcast by the **asm-break** command, to which primitive event ‘foo_ending is bound. When the inferior encounters (any member of) this second breakpoint (set) under the control of the **while—step** loop, Dalek will once again stack that loop, allowing control to return to the event realm so we can finish the job by recording ‘foos_return_value.

But rigorous adherence to such conventions might exact a high price from the user, since in some cases, the overhead (in terms of number of lines of debugging code) might exceed the cost of writing the essential debugging code. In addition, each additional line of debugging code would seem to increase the chances of introducing a bug. For either reason, adhering to coding conventions could then have the same chilling effect on the user’s motivation to explore different approaches that we have inveighed against throughout this work. If the disincentives induced by any proposed convention turn out to be truly formidable ones, then that proposal cannot be considered to offer a viable solution.

6.5. Language-level Support to Minimize Code Interference

It would not be terribly difficult to introduce some new syntax to incorporate the two major coding conventions shown in Figures 6-3 and 6-5 directly into Dalek’s language. For example, new commands to **while-step** and to **maybe-cont**, with the obvious semantics, would avoid the most prevalent sources of false negatives. Dalek’s implementation could maintain a global convenience variable, "$WHILE_STEP", that would be set by **while-step** upon entry and cleared by that command when it terminated. **maybe-cont** would consider the value of $WHILE_STEP in determining whether it should resume the inferior, or should instead exit and defer that action to a pending **while-step** loop.

For that matter, we could establish **maybe-cont** as a user-defined command with no changes to Dalek’s implementation. This technique would not work for **while-step**, because user-defined commands cannot take arguments. If instead, we wish to realize **while-step** as a user-defined function, this will require the user to pass its arguments as strings (to avoid immediate evaluation). The user can accomplish this by coding the **while-step**’s termination and recognition conditions, and its recognition action as three separate user-defined functions; he must then pass pointers to these debugger functions (Dalek *does* support them) as arguments to the **while-step** function. But this method makes even more work for the user! Thus a friendly realization of the proposed **while-step** command might require the implementation, in effect, to support string invocation of Dalek language commands. We might then choose to offer this capability to the user directly, as well, in the form of a new predefined function, **$eval**().

For the prevention of false positives, no new syntax would be necessary. Rather, we would ordain a default semantics: Any event raised as a result of an "artificial call" would be annulled before Dalek even started to interpret its command block. The implementation could be made flexible so that, if he chose, the user could explicitly override this default.

6.6. Managing 'Special Cases' of Interference

The previous sections pursued an unabashedly bottom-up strategy in tackling the problem of interfering command blocks. As the culmination of that approach, the proposed language extensions have undeniable merit, despite their lowly origins. Yet what may be their most serious flaw also arises from their ad hoc ancestry.

Although Dalek's purpose is to act as a vehicle for research into the fundamentals of debugging rather than to spoon-feed familiar techniques to novices via a menu-driven interface, we have no desire to be actively *hostile* to new users. Consider therefore, the plight of a novice Dalek user. Although it would be fairly easy for him to blindly follow coding guidelines stating how **while-step** and **maybe-cont** should be used, he would have to make considerable progress along the learning curve before he could fully understand the nature of the problem, and exactly *how* those guidelines solved the problem. Until this neophyte could overcome his dependence on those new language features and hand-code their precursors himself (much as we have done in the previous sections), he would lack the preparation to attack all *other*, special cases of code interference that the new language features were never intended to resolve.

To motivate this section, let us consider what may be the most recalcitrant "special case" of code interference: Suppose a **while—step** loop propels the inferior into a breakpoint whose commands initiate a different **while—step** loop. Clearly such a situation is just as prone to false negatives as is the code of Figure 6-2. Yet we previously agreed to hold the user blameless if two **while—step** loops, mediated by a breakpoint in the inferior, were to become dynamically nested. Perhaps at this point the reader may be wondering whether such situations should be of concern; i.e., would such situations arise routinely in practice? Unfortunately, such circumstances might well abound in practice, if Dalek were robust enough to handle them. Such dynamically nested "watch" loops might even be so prevalent that we would be unjustified in considering them a "special case"!

6.6.1. The Semantic Integrity of an Isolated Watch Loop

It will prove useful to classify watch loops into two types: those whose loop expressions and/or bodies contain any references *by name* to inferior entities, and those that do not. Clearly, the most straightforward way to initiate a loop for monitoring (by name) a local inferior variable would be from a breakpoint located in the corresponding inferior procedure; the watching loop should terminate at or before the time that same invocation of the inferior procedure exits. Of course, such local loops *must* propel the inferior with the **next** command, since a stride taken at the granularity of a single **step** could transfer execution into a different inferior procedure — and then immediately return control to Dalek before that inferior call had a chance to complete. As discussed in Section 5.4.1.1, by thus crossing a scope contour line, the referent of an inferior name can be altered, thereby compromising the semantic integrity of the debugger's watching loop. But, as was also pointed out there, if the watch loop refers to inferior entities *by name*, then even using **next** is not sufficient to guarantee that the debugger's watching loop will regain control at *only* those points in the debugging computation where the inferior's execution intersects a plane of constant scope in its contour diagram. This is because C allows unnamed blocks, in which new local variables can be declared, to be nested within any procedure. If a **while—next** loop were to propel the inferior into such a block, this would jeopardize its own referential integrity.

Now consider how best to monitor a global variable, by name. Although globals are at least well-defined throughout the extent of the inferior's execution — thus permitting a watch loop to employ the **step** command — it is apparent that in doing so, the global watcher too would jeopardize its own referential integrity. Nor would **next** be a safe alternative. Although its periodic checks on the global variable would be spaced much further apart in time, the watcher would still be susceptible to driving the inferior into a nameless, nested block, and then falling in after it. Should a newly-declared local variable there mask the watcher's view of a previously visible global inferior variable, this could compromise the watch loop's programmatic mission.

For these reasons, we shall designate any **while** loop that both resumes the inferior's execution, and that uses an inferior name, as a *Semantically Unstable*, or "SU" loop.[3] But we need not eschew the use of watch loops altogether, merely because some of their variants might exhibit unstable semantics during the course of the debugging computation. On the contrary, the discussion in Section 5.4.1.1 pertaining to Figure 5-2 explained a safe method for employing a loop to watch a local inferior variable (at least, so long as that local was not a *register* class variable). Fortunately, since global variables always maintain (virtually) a constant "home address", the method we described there could safely be applied to watch any global variable, as well. Our previous discussion also indicated that, by using an address specification rather than a name when referring programmatically to an inferior object, it was even possible to watch a local variable when the inferior was not currently executing within the scope of its corresponding procedure. Indeed, certain debugging situations may require us to watch a local variable during just such

[3] Any phonetic resemblance to actual Fortran loops, living or dead, is purely coincidental, as are any semantic parallels with the behavioral specification of certain legal (ne computer) firms.

periods — while its owning procedure lies dormant, awaiting the return from a call it has made to another inferior procedure. Perhaps this is because the original procedure has exported the address of its local to the newly invoked procedure, and we must watch for any changes caused by the latter. Alternatively, there may be an errant pointer somewhere in an unknown routine that is corrupting the address at which our local variable resides on the inferior's runtime stack.

6.6.2. A 'Loosely-Bound' Watch Loop Partitions the Logical Timeline

There are two points to this discussion. One is obviously that *if* a user chooses to employ SU watch loops, he should be cognizant of their potential hazards. The other point regards the further classification of the remaining form of watch loops — those that are semantically *stable*. Because a loop watching a local variable might wish to use the **step** command for extra vigilance, and because a loop watching a global variable might choose to employ the **next** command for efficiency, it is not necessarily true that the stride length of a watch loop is inversely proportional to the scope of an inferior object it has been assigned to monitor. Besides, a single watch loop might contain code to monitor both global, as well as local variables.

Nevertheless, when we speak of a "local" watch, we shall generally assume that it is primarily concerned with local variables, and that it employs a **next** command to propel the inferior through the course of its execution. In so doing, it is clear that the local watch would effectively choose to sleep much of the time, awakening only at isolated "points" (from the perspective of the inferior's logical timeline) to monitor the inferior's state. Similarly, a "global" watch that employs the **step** command to propel the inferior would also sleep some of the time, although it would need to awaken far more frequently to monitor the inferior's state — namely at all those points on the inferior's logical timeline at which its program counter register was positioned at the beginning of a line of source code. From the inferior's point of view, a solitary watch imposes its own characteristic partition on the inferior's logical timeline. Thus the periods of alertness of a solitary watch loop can be viewed as dividing the course of the overall debugging computation into a series of intervals, wherein the command block for that particular watch "executes" in strict alternation with intervals in which the debugging computation is dominated by the inferior process.

We abstract out certain details in order to focus on what seems to be a very useful concept, namely the *extent* of a watch loop. To make this concept precise, we must first sharpen our terminology a bit: An *activation* of a watch loop will correspond to Dalek's interpretation entering the top of such a loop. This must be distinguished from *activity*, in which the watch loop will engage every time it wakes up in response to a transfer of control to it from the inferior. (Isn't natural language wonderful?) Note that, unfortunately, there is not a direct correspondence between an activation of a watch loop and Dalek beginning to interpret the command block in which it resides. For example, one command block could contain two watch loops in series, or they might even be nested. We will avoid such constructs in the present discussion, since matters are already complicated enough. We will, however, also use *activation* to refer to an instance of Dalek beginning to interpret a more straightforward command block, such as that of a breakpoint that does *not* contain a watch loop, and whose *extent* is a single point on the inferior's logical timeline.

We are interested in the *endpoints* of a particular activation. That is, given the logical timeline representing the inferior's execution, what are the earliest and latest points at which activity due to one activation of a watch loop appears? (We will assume the inferior eventually terminates.) Of course, viewed as a debugging computation rather than merely from the inferior's point of view, these endpoints, as well as any intermediate "points", expand to become some of the intervals we spoke of earlier. But we are especially concerned with the length of the strides an activation of a watch loop takes in traversing the segment between its endpoints. Although it may *intend* to take a stride of a certain length, the *actual* stride may be considerably different — in effect, it may be "shoved" due to interference from another activation. This is exactly what would happen to the watch loop of Figure 6-1. In that case, what should have been its terminal endpoint would be shifted forward considerably in time — perhaps all the way to the inferior's terminus; the intervals into which it would otherwise have decomposed the debugging computation would all be collapsed into that same point. Whenever the characteristic interval structure that would otherwise be imposed on the debugging computation by an activation of a solitary, isolated watch is disturbed, we will say that a *collision* has occurred between two (or more) of the debugging coroutines. As we indicated in the discussion regarding Figure 6-1, such collisions can easily give rise to false negatives. It is also possible for collisions to cause false positives, since if a particular watch were ever awakened too late *or* too early, it might see inferior states with which its logic (or that of any events it might raise) was never designed to cope.

Recall that in Section 3.2, as a first cut, we divided primitive events into two classes, free-floating and bound. In essence, we described a mapping between the command blocks in Dalek that contain an **event raise** command, and regions of source (or object) code of the inferior. This enabled us conceptually to localize an occurrence of a bound event in the "textual space" of the inferior's source code, and in the virtual address space of its object code.

Now consider a "freestanding" watch loop entered by the user at Dalek's top interactive command level as soon as the inferior begins executing. If it raises an event, then unless that free-floating event instantiates its attributes using calls to **$func**(), **$file**(), or **$line**(), the best we could do with a particular free-floating occurrence would be to localize it in *time*, since Dalek automatically assigns a logical time stamp to event instances as they occur. Even if a free-floating event did employ, e.g., **$func**(), in general, we might expect its occurrences to be distributed, seemingly randomly, throughout the inferior's textual space.

Occurrences of an event that is bound to a *set* of broadcast breakpoints would presumably be somewhat more amenable to "conceptual localization", although they would clearly not, in general, be confined to a single point in the source code. Although our simplistic dichotomy between free-floating and bound primitive events is not an absolute one, we nevertheless wish to maintain it as a convenient fiction for the specific situations we have just described.

Now consider a local watch that does *not* begin life as a freestanding loop at Dalek's top level, but is instead initiated when the inferior hits a breakpoint constituting one *endpoint* of the segment that determines the *extent* of the watch. Although such a watch loop might propel the inferior all the way to its termination, it is also possible that the watch will encounter its own, personal endpoint long before the inferior finishes executing. We will use the term, *loosely-bound*, to designate exactly those watches whose duration is *not* coextensive with that of the inferior's execution. Any event that may be raised within such a watch loop will inherit its loosely-bound status. Although one purpose of loosely-bound watch loops is to achieve better realtime performance than would be possible with entirely free-floating watches, note that another purpose of loosely-bound watches is that they allow the user to "conceptually localize" an event not within a contiguous region of source code, but rather within certain segments of the inferior's logical timeline. That is, the event potentially is "bound" to a limited *sequence* of states on the inferior's logical timeline. In discriminating *bound*, *loosely-bound*, and *free-floating*, the terminology is intended to be helpful; clearly, these distinctions are not absolute.

6.6.3. Tradeoffs between Semantic Integrity and Performance

So how do we profit from all this terminological baggage? First, note that it is impossible for two free-floating watches to collide during the inferior's execution. Since, by definition, both begin life at the same moment, and share the same extent, they must be the same loop. The user merely would have entered a freestanding **while** loop in which were embedded two different conditional statements, e.g., if x has changed then raise one event, and if y has changed, raise another. Thus, in the absence of loosely-bound watches, the *only* type of collision that can occur is between a free-floating watch and a block of breakpoint commands that causes the inferior to resume through some mechanism *other* than that of initiating a new watch loop. By far the most common way to resume an inferior is to give it free rein by using the **cont** command, yet Figure 6-3 illustrates an almost effortless coding convention to prevent collisions in those situations. A command block that resumes the inferior in any other way must intend for the inferior to traverse some more limited segment of its timeline before returning control to that command block.

If the command block contained some command — *inside* a **while** loop — that resumed the inferior, then by virtue of its form, this would comprise a loosely-bound watch loop. But by assumption, that cannot be the case. Alternatively, that command block may propel the inferior across some segment of its timeline by means of commands that are either unconditional or are guarded by **if** statements. Therefore the original command block must be composed of two smaller blocks, a "car" and a "cdr", separated by whatever command first resumes the inferior. In that case, it can be split into those two smaller blocks, each of which can be bound to two different (sets of) breakpoints. The interval between these two endpoints can be traversed by appending a properly guarded **cont** command to the first of the smaller command blocks. Thus, proceeding by induction on the number of "resumption" commands in the original block, we can convert it into a series of breakpoint command blocks, each of which is guaranteed to be collision-free since it concludes with a guarded **cont** command. Therefore, even if the segments determined by two different breakpoint series were to overlap on the inferior's timeline, no collisions would ensue. We demonstrated one special case of this "serialization" method in Figure 6-7, where we avoided the necessity of calling **$quiet_finish**() programmatically, within an event activated from a breakpoint.

It might be either inconvenient or cumbersome to perform this serialization process statically, before the inferior is run. In such cases, the technique can be applied dynamically — one breakpoint's code can (conditionally) set its successor and establish a command block for it, just before resuming the inferior as its final act. If the user wishes to delete these "induced" breakpoints once the inferior encounters the endpoint of the original segment,[4] this can easily be accomplished via breakpoint "names". The name denoting each new breakpoint (set) in the series can be appended to a global convenience variable of type string, dedicated for use by that particular series.

Consequently, to prevent collisions it will be sufficient that we not employ loosely-bound watch loops. But in that case, how could we watch a local variable? One method might be to employ a single, freestanding **while** loop, which, after every step, would first compare the name of the procedure whose local variable we wish to monitor, with the value returned by a call to **$func**(). While this might not be nearly as efficient or convenient as initiating a loosely-bound watch from a breakpoint inserted in the target procedure, it might be worthwhile since it ensures the absence of collisions, and therefore prevents false negatives from corrupting the semantic integrity of the event history.

6.6.3.1. Counting *Strides* is a Universal Performance Metric

But it would be extremely *inefficient* to single-step the inferior throughout its lifetime in order to watch for a condition that is only well-defined within one procedure that may be called only once. In fact, on *any* operating system running on *any* hardware (including even multiprocessors and shared memory), if the debugger and the inferior execute as two different processes, then it is a *logical necessity* that every transfer of control between them entails a cost of at least two context switches. Although details of dif-

---

[4] For situations such as described in Section 4.1.1 (i.e., Figure 4-3), it would be absolutely vital to delete the induced breakpoints.

ferent architectures and operating systems would cause the *realtime* cost of such context switches to vary considerably, nevertheless, in terms of the "local currency" on any particular system, a context switch would still be relatively expensive. Of course, no context switches would be required if the same process "debugged itself" by using specially linked-in debugging routines, but any such scheme could profoundly (and unintentionally) disturb the inferior's natural course. In addition, the inferior, through errant pointers, could corrupt the recorded observations of its actual behavior.

So if we require semantic integrity, then we should use two processes, but — for practical purposes — should strive to minimize the number of control transfers between them. We will assume that for the debugger to read or write in the inferior's address space is relatively cheap, since, in contrast to a control transfer, it is *not* logically necessary for the operating system to perform any context switching. Thus an extremely practical measure of performance is the number of *strides* taken by the debugger to reach a given point on the inferior's logical timeline. For example, traversing a given interval on the inferior's logical timeline via the **step** command would, in general, require far more strides than traversing that same interval using the **next** command. Of course, the cheapest way of all to traverse that interval would be to set a breakpoint at its endpoint, and then to resume the inferior with the **cont** command. This would cover the interval in a single stride. However, we must not ignore the user's pragmatic objectives in our haste to propel the inferior to a given point on its logical timeline. After all, in many phases of debugging the user desires not only to reach a given point in the inferior's computation, but also to collect certain information along the way, programmatically. He may even wish to **pause** to smell the flowers for himself.

Now because we are interested in the general principles of debugging, i.e., those not solely applicable to a single machine or operating system, we will employ the number of *strides* as the sole measure of performance. Although it is true that many implementations of UNIX require two context switches to read an inferior's address space, there are other implementations that support the "/proc" virtual filesystem, making read and write accesses so cheap that essentially, we need not even consider them. Thus, counting strides gives a valuable performance measure within a portion of the UNIX community, in particular.

Having established that measure, we realize that to employ a free-floating watch to replace a loosely-bound one would, in general, impose an intolerable performance penalty on the debugging activity, on *any* operating system. Thus it would seem that the user's desire for performance is in direct conflict with his desire for preserving the semantic integrity of his debugging activity. If we cannot guarantee that collisions will be prevented, is there, perhaps, some way they can successfully be *managed*?

6.6.4. Managing Collisions via the Interrupt Priority Model

Rather than relying solely on ad hoc coding conventions disguised as linguistic features, a radically different approach to the problem of code interference would be to candidly admit from the start that the user's debugging code must operate in a complex environment where each command block potentially executes as a re-entrant coroutine. While this realization might initially intimidate novice users, the majority of them should already be equipped with the necessary conceptual machinery, although they may not realize it. The proposal in this section is not to force novices to grapple with the semantics of general, re-entrant coroutines, but rather to take advantage of the semantic parallels between unexpected breakpoints and asynchronous interrupts, and to exploit the resulting analogy between breakpoint command blocks and interrupt service routines. Along with the traditional notions of specific interrupt priorities and of a global interrupt fence, this model offers a framework that is powerful enough to encompass the ad hoc approaches of the previous sections and that can easily be grasped by most novices.

Much could be accomplished with a simple interrupt model. Suppose the user is given the opportunity to assign every breakpoint command block a priority level. If block *A* somehow **step**s the inferior into a breakpoint having command block *B*, the priority levels of the two blocks could determine whether *A* should be stacked so that Dalek can begin to interpret *B*, or whether *B* should be ignored in this particular case. Only if the two colliding blocks had the *same* priority level *might* Dalek need to interactively query the user for guidance.

Suppose two *local* watch loops with the same priority were to "collide". By definition, local watches must somehow contain the necessary termination logic so that they do not outlive their associated inferior procedure’s current invocation. This could be accomplished in several different ways; the specific coding style chosen by the user is not germane to the present discussion. What is relevant is that because the recognition condition of either of the two watches is only well defined when the inferior is executing within that watch’s associated inferior procedure, it is never necessary (or advisable!) to check *both* watch conditions immediately after Dalek interprets a particular **next** command.[5] Thus the *latest* local watch encountered should supersede the older local watch, and Dalek should automatically pop its command block stack and resume the older watch once execution in the inferior returned to its associated procedure. Thus by assigning all local watches the *same* priority level, and by somehow identifying them to Dalek as *being* local, "collisions" between local watches could easily be resolved without bothering the user. In fact, such encounters between local watches propelled by the **next** command would *not* satisfy our formal definition of "collision". This is because both endpoints of the innermost watch would lie strictly between two adjacent periods of activity of the outermost watch. Although the inner watch would create a *sub*-partition of the inferior's logical timeline, it would *not* alter the natural structure imposed by its outer cousin.

In contrast, the situation would be quite different if two *global* watch loops, loosely bound to breakpoints in the inferior, were to collide. Then not only would *both* watch conditions need to be checked after every **step**, but in addition, both watch loops might remain active until the inferior process finished executing.

Thus it would violate the user's intention for either watch to supersede the other. Therefore, both global watches should be identified to Dalek as *being* global, and should be assigned identical priority levels by the user. Should a collision between them occur, Dalek would need to notify the user — unlike its response to "ties" between two *local* watches. In such a situation, the only solution would be for the two watch conditions to somehow be combined into a single, unified watch loop. The loop combination could be

---

[5] However, both watch conditions *should* be checked "simultaneously" if the two watch loops are loosely-bound to *overlapping* sections of the same inferior procedure. The semantics of this case are equivalent to those of a collision between two global watch loops.

accomplished manually by the user, if and when Dalek notified him of the collision.

If a *local* watch and a *global* watch were to collide, the user might again choose to commingle the two loops. But in this case, he might want to include logic that would bypass the local watch condition whenever the inferior process was not executing within that watch's associated inferior procedure. The alternative would be to intercede manually to "de-combine" the watch loop when the current inferior procedure exited — not a pleasant prospect when one considers how often the user might need to perform that sequence.

Yet if this practice were followed, one can easily imagine a global watch loop that began life as a short block that originally checked only one condition: If no de-combining were ever performed, it could eventually expand to include hundreds of other watch clauses, the vast majority of which would be inactive during an arbitrary pass through the watch loop. Although this would not necessarily present a danger, it could not be considered either elegant or highly efficient. Thus even the interrupt priority model does not seem to offer an ideal solution.

Still, having achieved such a promising result from our top-down approach, is there perhaps some way we could evaluate the effectiveness of the interrupt priority model in managing collisions between free-floating and/or loosely-bound events, without engaging in another strenuous round of development work to modify Dalek's implementation? After all, since neither of our two ad hoc coding conventions was designed to prevent false negatives under these circumstances, we do not seem to have all that much choice. Besides, it remains to be seen just how prevalent such collisions would actually be in practical debugging situations. As with most issues in debugging, the problem statement is necessarily a vague one: "Is the interrupt priority model the most convenient for — and least confusing to — J. Random User?" For this reason, only experience can ultimately judge the success of any particular strategy.

Nevertheless, we consider it an auspicious omen that one root of the problem — Dalek's full programmability — itself provides a foundation for assessing the adequacy of the proposed solution: Dalek's language, being general-purpose, allows us to *simulate* the interrupt priority model without requiring any further modifications to Dalek's own implementation!

6.6.4.1. A User-level Implementation of the Interrupt Priority Model

Thus Figure 6-8 shows what seems to be yet another coding convention, this time, to guard a generic command block within logic that manually implements the interrupt priority model. The coding convention utilizes two global convenience variables, $PRIORITY and $WATCH, representing respectively the current height of the interrupt fence, and the *type* of watch loop currently propelling the inferior (i.e., local, global, or neither). It also employs local variables $my_priority, $my_watch, $Old_PRIORITY, and $Old_WATCH, the latter two for storing the previous global state when the priority level of a newly activated command block supersedes that at which the dynamically preceding command block was running.

This model provides a mechanism that permits the unification of several previously distinct semantic situations. For example, breakpoints themselves (as opposed to their associated command blocks) were formerly *binary* entities — a breakpoint was either **enabled** (i.e., inserted in the inferior's codespace) or **disabled**. However, if the user has

```
break foo
commands
  {            # Start of "interrupt service routine" to check priorities.
  local $my_priority      # Priority level user assigns to this command block.
  local $my_watch         # If a watch, is this command block local or global?
  local $Old_PRIORITY, $Old_WATCH     # Store former values of globals.
  set $my_priority = 73              # Assign this block a priority.
  set $my_watch = "local"            # Or "global" or "neither".

  if ( $my_priority < $PRIORITY )       # Do nothing on our own behalf!
  else if ( $my_priority == $PRIORITY  &&  \
              ( $my_watch == "global"  ||  $WATCH == "global" ) )
    # Notify user and pause, allowing him manually to combine the two loops,
    # or temporarily to boost $my_priority, or whatever.
  else if ( $my_priority > $PRIORITY ) # Hurdle global interrupt fence, or
    # this may be a collision between two local watches.  Either way, go for it!
    set $Old_PRIORITY = $PRIORITY
    set $PRIORITY = $my_priority     # Since we might tunnel into another
    set $Old_WATCH = $WATCH      # block, or while—step below.
      # Now insert the normal body for our command block:
      # It may possibly  set $WATCH = $my_watch.
        . . .   # Main body of command block.
    # Restore former global state, once control returns here:
    set $WATCH = $Old_WATCH
    set $PRIORITY = $Old_PRIORITY
  endif
  }            # End of interrupt priority coding convention.
  # Finally, close by subsuming an earlier coding convention:
  if ( $WATCH )
    set exit 1
  endif     # I.e.,  if (not propelled by a while—step loop) then . . .
  cont      # (Avoids false negatives, and tail recursion too.)
end
```

Fig. 6-8 Coding convention to implement the interrupt priority model.

consistently employed the proposed coding convention, guarding every breakpoint command block with logic to compare its own priority with the current global interrupt fence, then he can issue a command, perhaps interactively while **pause**d at an arbitrary point in the debugging computation, to "**set** $PRIORITY = $Insurmountable_Fence". This command would effectively **disable** all breakpoints! Issuing commands to set the global interrupt fence progressively lower would have the effect of enabling more and more breakpoints. Although such a maneuver, on the face of it, might seem to have no place whatsoever in a *structured* debugging methodology, it makes a good deal of sense when one considers that every member of a set of *broadcast* breakpoints will necessarily begin an activation with the same initial value of $my_priority. Thus any coding convention that implements the interrupt priority model provides another mechanism, completely independent of that discussed in Section 2.2.3, for selectively enabling or disabling a *class* of breakpoints. Recall that the technique of Section 2.2.3 permits the user to "name" a set of breakpoints whose membership is independent of the broadcast (or singleton) sets in which its members reside. In contrast, manual or programmatic adjustment of the global interrupt fence is incapable of discriminating at any finer level of granularity than that of the current membership of the broadcast sets. Thus adjusting the height of the interrupt fence is actually a more structured (and therefore less powerful) method, since it forces all residents of a broadcast set to be enabled or disabled alike. It also enforces a total ordering on the broadcast (and singleton) sets, whereas the breakpoint "naming" mechanism shown (and perhaps abused) in Figure 2-7 imposes no such corresponding structure.

Any coding convention that implements the interrupt priority model can also easily subsume the previous ad hoc coding convention for preventing false positives as well. Rather than guarding an event's command block with code to check the ad hoc global variable $artificial_call, as in Figure 6-5, it is more than sufficient that all "artificial callers" temporarily elevate the global $PRIORITY fence to an insurmountable level, thus preventing any spurious event from ever being raised in the first place. Such a technique is semantically akin to a CPU running with interrupts disabled, a concept that should be familiar even to novice users of a debugger.

One would expect that, by default, the $PRIORITY of Dalek's top interactive command level should be zero, thus allowing any command block activated during the course of execution to take precedence over it. In addition, certain command blocks, e.g., user-defined functions that are entirely "passive" — guaranteed never *directly* to resume the inferior — do not need to be guarded by any $my_priority level of their own, but instead will automatically inherit that of their caller, on whose behalf they operate. For such routines, the user obviously need not include the "**if** ($WATCH)" segment at the tail of Figure 6-8 either. Of course, any block that might directly raise an event must be guarded, since although it might not resume the inferior, it must still protect the integrity of the event history from "artificial callers".

6.6.5. Comparison of Watches with the RAIDE/Dbx 'when' Construct

By this point, experienced dbx users may be wondering whether we are not just beating a dead horse. Perhaps the expressiveness of Dalek's current language constructs, specifically, variants of the **while—step** loop, is simply not adequate for what we want to do. Why not jettison the whole notion of hooking loosely-bound events onto the inferior by means of watch loops, and instead adopt a method used by dbx to implement its counterpart of "free-floating events"? For example, some versions of dbx allow the user to employ a syntax such as

when (*foobar* < *xbar*) **do** {**cmd; cmds** }
or
trace (*foobar*) **in** *sub_routine*

Could Dalek provide a similar syntax for the first, "free-floating" example, and implement it by invisibly **step**ping the inferior, checking each **when** expression after each step, and conditionally performing the associated actions? For that matter, the second example of dbx code certainly *looks* a lot like a loosely-bound watch! That is, the user is telling dbx that it does *not* need to single-step anywhere *except* in procedure *sub_routine*. Unfortunately, for whatever reasons, the version of dbx on Sun3 machines that supports both the constructs shown above chooses to implement *both* by single-stepping at the machine instruction level throughout the entire course of the inferior's execution. This is true even if *only* the second command is issued.

In principle it would be entirely feasible for Dalek to employ the **when** syntax, but there are important practical concerns dealing with efficiency — as well as semantics — to be considered. We have used the terminology "loosely-bound" to describe how a watch loop might be active over only limited regions of the inferior's logical timeline. If the inferior were executing in other regions, it would be allowed to proceed at full speed, unhampered by the necessity to continually undergo operating system context switches so the debugger process could check certain conditions. Thus, in situations where an inferior would need to spend only a small fraction of its CPU-time under the control of watch loops, Dalek's current method would be far more efficient than that presently implemented for dbx, since it would require fewer strides to watch the same conditions.

In a situation where hundreds of such loops were loosely-bound to various regions of the inferior’s logical timeline, we could wind up invisibly **step**ping the inferior throughout virtually every region of its execution.[6] Thus it appears that at the asymptotic limit, the efficiencies of the two methods should converge, and the semantic integrity should generally be much better under dbx. If there were no way to guarantee semantic integrity when loosely-bound watches were employed, that would be very unfortunate, since the efficiency advantage of an individual loosely-bound watch translates into another very desirable, *collective* property: As the asymptotic limit is approached — that is, as more watches are loosely-bound to various segments of the inferior’s logical timeline — the overall performance of the debugging computation will degrade gracefully. This is in sharp contrast to the "all or nothing" style of performance degradation experienced when even a single free-floating watch is initiated.

It would seem that, at least in some debugging situations, dbx’s method might be significantly superior to loosely-bound watch loops. The syntax furnished by dbx for this particular construct is undeniably attractive. We might consider extending the **when** syntax with the addition of two new clauses to explicitly describe the endpoints of the region(s) on the inferior’s logical timeline to which that **when** condition should be loosely-bound.

So far, this sounds like an attractive proposal. But would this really accomplish anything? Recall that in the interrupt priority model, when collisions occurred, it was the

[6] Not to mention tripping over ourselves and causing meta-Heisenberg effects.

user’s responsibility manually to commingle (and subsequently de-combine) watch conditions. If we merely were to implement **when** as a new syntactic form, leaving the user with the responsibility for managing collisions, we would have succeeded only in disguising the problem. Instead, to *solve* the problem, the burden of performing the commingling and de-combining chores somehow must be shouldered by the language implementation.

Note that to be as powerful as a loosely-bound watch loop, one of the proposed new clauses on the **when** construct must itself be capable of expressing an *arbitrary* condition, not merely something as simple as, e.g., "de-activate me when you exit *sub_routine*, or when execution reaches line 210". Thus, to express an arbitrary watch termination condition would require *another* **when** statement, e.g., "quit watching if ($x > y$) **when** ($p <= q$)".

Thus there are two questions: whether a syntax extended thusly could still be almost as palatable and user-friendly as the unadorned **when** construct, and whether the language implementation required to achieve it might be enormously complex and correspondingly prone to bugs itself.

6.7. A Synthesis: Semantic Integrity with Efficient Performance

So far, it has seemed that, for reasons of practical performance, the loosely-bound watch loop may be a necessary evil. It is a powerful construct, but seems to be correspondingly risky. When the extents of two such loops overlap on the inferior’s logical timeline, the result is a transfer of control from one of the user’s coroutines to another via "tunneling" through the inferior. Some such control transfers are benign because they merely add additional fine structure to what would otherwise be the natural partition imposed on the inferior's logical timeline by the first watch loop. However, other encounters, those we technically term *collisions*, distort a prior partition in the act of imposing their own structure.

Much earlier, we proposed dispensing with loosely-bound watch loops altogether in favor of a single, free-floating loop. The advantage was a guarantee of semantic integrity. The disadvantage was that it would perform so slowly that one would be reluctant to use it. But we were too hasty in dismissing that idea. In fact, there is a way to employ a single, free-floating loop, yet optimize its performance to match that of an *arbitrary* collection of loosely-bound watch loops. As long as each watch is semantically independent of the others, we can, in effect, decompose the inferior's logical timeline so that it reflects a *union* of all those partitions that would be imposed by each individual watch loop acting in isolation.

In keeping with the elegance of this new method, we call it "TOPDOG". Although it is possible to implement TOPDOG and its auxiliary code entirely at the user level, the code would be very dense and would need to make extensive use of pointers to user-defined functions. However, our primary objections to a user-level implementation are that TOPDOG requires several *list* structures, and the new "pseudo-watches" need *static* local convenience variables: Currently, lists and static local variables can *only* be realized in Dalek through the queues of a high-level event. We feel that to employ events for such purposes would be an abuse of their intended semantics. Therefore, as an alternative, we employ several new predefined functions that manipulate implicit lists on behalf of TOPDOG, and assume Dalek supports the more conventional form of static local variables.

The construction, as shown in Figure 6-9, employs a global convenience variable, $watchpoints_active, to act as a counting semaphore. The key idea is that TOPDOG is the *only* one of the user's coroutines authorized to propel the inferior along its course. Whenever, using the old method, a loosely-bound watch would have been activated, its counterpart — a pseudo-watch — instead "registers" certain information, including its preferred stride, with TOPDOG. TOPDOG acts as a central decision maker. It always takes the *shortest* stride among all that have been requested. In so doing, it ensures that no false negatives — caused by a watch "oversleeping" — can occur. After every stride, TOPDOG is responsible for awakening exactly those pseudo-watches that would ordinarily have awakened at that point on the inferior's timeline if they were the only activated watch. By maintaining this invariant, TOPDOG can ensure that no false positives will be generated either, thus preserving the semantic integrity of a debugging computation that would otherwise have employed an arbitrary collection of (possibly colliding) loosely-bound watches.

Whenever one of these new pseudo-watches terminates, it must notify TOPDOG as well. Thus, during intervals when nothing is being watched (or more generally, during intervals in which the number of hardware watchpoint registers is sufficient to do the job), TOPDOG will propel the inferior with the longest possible strides, that is, it will use the **cont** command. Therefore, TOPDOG can achieve semantic integrity using no more strides than would a collection of watch loops managed by the user manually via the interrupt priority model.

New pseudo-watches could be registered with TOPDOG interactively (from **pause**d mode), by the code at breakpoints, and by the code of primitive or high-level events. In

```
# We construct a "virtual machine" to run near Dalek's top interactive level:
while ( $running )

   if ( $watchpoints_active  <=  $watchpoint_resources )
      cont    # Allow the inferior to run free, until it either hits a
              # breakpoint or accesses an address watched by hardware.
   else
      # Hardware resources exceeded; must watch via software.
      if  ( $list_length ($machine_step_watchlist) )
         stepi
      else if  ( $list_length ($source_step_watchlist) )
         step
      else
         next
      endif
   endif
   set  $logical_ticks++

   # Always awaken any and all watchers at the machine instruction level:
   call $awaken_watch_list ($machine_step_watchlist)

   if  ( $source_line_boundary ($pc) )
      # If currently positioned at beginning of a new source line,
      # then awaken any and all watchers at the source line level:
      call $awaken_watch_list ($source_step_watchlist)

      # Conditionally awaken some watchers whose preferred
      # stride is next.  Such watches are only awakened if the
      # inferior's current frame matches that at which the watch originated:
      call $awaken_some_on_watch_list ($source_next_watchlist)
   endif

endwhile
```

Fig. 6-9 The TOPDOG approach: an elegant alternative to loosely-bound watches

Figure 6-10, we present one such example that uses a hypothetical new command, **begin-watch**, to register a pseudo-watch with TOPDOG. Since this pseudo-watch

---

```
begin-watch
  static $self_id, $old_x_value
  begin-initial
    set  $old_x_value = x   # Watch a local variable.
    set  $self_id = $register_my_stride ($source_next_watchlist, $fp)
    # We will need $self_id later to de-activate ourself when appropriate.

    # We show how a watch might claim a hardware watchpoint register,
    # but generally, those precious resources should be used only by
    # watches that would require step-sized or smaller strides:
    if ( $watchpoints_active  <  $watchpoint_resources )
      call  $reserve_watchpoint_resource ($self_id, &x)
      set  $watchpoints_active++
    endif
  end-initial
  begin-body
    # TOPDOG will awaken this body when inferior reaches any source line
    # boundary, if its then-current frame is no higher than the value of $fp we
    # gave when we initially registered our stride on $source_next_watchlist.
    if ( $func(0) != "original_inferior_procedure" )
      # In general, our termination condition above could be arbitrary.
      call $terminate_watch ($self_id)
    else if ( $old_x_value != x )
      event raise ‘x_changed (x, $logical_ticks)
      set  $old_x_value = x
    endif
  end-body
end-watch
```

Fig. 6-10 Registering a newly ‘activated’ pseudo-watch with TOPDOG.

---

prefers a **next**-sized stride, it must also inform TOPDOG in which frame of the inferior's runtime stack it is "anchored". This way, TOPDOG will know not to awaken it (i.e., activate its body) by mistake when the inferior is executing in more recently invoked frames.

If Dalek were run on a machine based, e.g., on the Intel 80386, TOPDOG would be able to watch up to four locations cheaply, without reverting to single-step mode, since that chip has four watchpoint registers to support bus-snooping. Most architectures, though, have zero $watchpoint_resources. On such machines, TOPDOG would be forced into single-step mode throughout the entire course of the debugging computation, if there were even one free-floating primitive event.

What the TOPDOG construction of Figure 6-9 achieves is essentially a highly efficient sequence of strides to cover the inferior's timeline, without *any* possibility of collisions. Thus we can avoid semantic interference entirely, so long as the user employs address specifications for inferior objects rather than naming them directly. (Again, we are ignoring register variables and compiler optimizations.) Since the method of address specification we displayed in Figure 5-4 was rather cumbersome, Dalek could be enhanced to understand *fully qualified* names instead, e.g., *@foo@bar* might refer to a variable local to the most recent invocation of procedure *foo* on the inferior's runtime stack. In general, *@(N)foo@bar* would denote whatever the name *bar* would denote in the context of the N^{th} most recent invocation of procedure *foo*. For convenience, perhaps *@(-M)foo@bar* could refer to *bar* as seen in the context of the M^{th} *oldest* invocation of procedure *foo* on the inferior's runtime stack.

Since the inferior is a compiled beast, it cannot be run on a virtual machine that responds favorably to debugging requests that certain addresses be watched. Nevertheless, TOPDOG simulates just such a machine, *without* necessarily incurring the cost of two UNIX context switches every time the inferior process executes a machine instruction. We pay for only as much as we need — namely, peace of mind that certain locations are being watched over *exactly* those intervals in the inferior's logical timeline that are of concern to the user. In this respect, the functionality achieved by our construction resembles that achieved by the technique of *selective interpretation* [Chas87], although the overhead penalty we suffer while in "selective interpretation" mode (due to the UNIX context switches) probably makes our construction run an order of magnitude slower than would a real interpreter that operated in the same address space as the inferior process. Although there are certain similarities between TOPDOG's pseudo-watches and Saber-C's ability to run an inferior composed of a *mixture* of compiled and interpreted code, there are important differences as well. Unlike TOPDOG and [Chas87], Saber-C does not presently allow a user to dynamically alter the mixture; instead he must re-start the inferior if he wishes a particular routine to be, e.g., compiled rather than interpreted. Another important distinction is that currently, Saber-C only detects references to watched locations that are initiated from within a region of interpreted code. Thus, Saber-C's equivalent of watches could be viewed conceptually as being "bound" strictly to regions of source text, rather than to actual temporal intervals on the inferior's logical timeline.

6.7.1. Can TOPDOG Achieve a Tight Lower Bound on Performance?

We have shown that through the TOPDOG mechanism, we can perform a debugging computation *semantically equivalent* to any which the user intended to accomplish by employing a set of loosely-bound watch loops, and *using exactly the same number of strides*. By semantically equivalent, we mean that for a given (repeatable) inferior program and input data, we could produce an event history identical to that generated if it were instead monitored using loosely-bound watch loops (presumably the user would employ the interrupt priority model to manage any collisions).

It might seem that TOPDOG, given a particular debugging objective specified by the user, should be able to achieve it, and to do so in the least *possible* number of strides. A simple adversary argument should suffice to show that, given an exact debugging objective, e.g., "Bring the inferior to point t_{237} on its logical timeline, and watch variable x during interval A", then any algorithm that took *fewer* strides would necessarily have to take *longer* strides, and thus could fail to notice the exact time when x was altered, or might even fail to notice that it even *was* altered.

Then such an argument would indicate that no debugger using any algorithm could achieve better performance than that achieved by Dalek using TOPDOG.[7] Unfortunately, there are two flaws in that reasoning. In fact, "selective interpretation" as practiced in [Chas87], if it were upgraded to support events, could achieve the same results on any

---

[7] In certain cases, intelligent analysis of the inferior might be able to determine, e.g., that because x can only be changed at line 128, we could achieve the debugging objective faster by setting a breakpoint at that line, rather than by single-stepping over even a limited interval of the inferior's timeline. For the moment, let us accept the implicit assumption that for whatever reasons, it is either not possible or not worthwhile to perform such intelligent analysis.

debugging problem as TOPDOG in *zero* "strides"! That is because, even though their method employs two different processes, "selective interpretation" in effect *simulates* the behavior of the inferior without ever transferring control to it. Although their method would require numerous reads and writes to the inferior's address space, as well as a significant amount of computation for the simulation, it would not technically require any context switches except when entering or leaving selective interpretation mode.

Thus, in cases where such "simulation" is employed, we would have to revise our definition of the stride as a performance metric by estimating, for a particular architecture and selective interpretation method, how many selectively interpreted machine instructions would incur the same realtime cost as an actual context switch. If selective interpretation were TOPDOG's only competition, then the performance advantage of loosely-bound watch loops and of the TOPDOG construction would still translate into optimal performance on *any* debugger, including those employing selective interpretation. One would first make use of all available hardware watchpoint registers, then typically use **next**-sized strides wherever possible. In general, one would only resort to selective interpretation in order to avoid taking **step**-sized strides in software.[8]

Thus, having already precluded intelligent analysis of the inferior, then once the performance metric is extended to become a realistic measure for schemes employing selective interpretation as well, it would seem that a minor variant of TOPDOG will indeed achieve a tight lower bound on the limits of possible performance. But even with those two caveats, such a claim would still not be justified, because we have not ruled out *unin-*

---

[8] Note that in some sections of an inferior, **next** would be just as slow as **step**, e.g., within the body of a computationally-intensive loop that did not call any other procedures.

telligent analysis of the inferior. Specifically, an entirely different method of "watching" for alterations to the contents of some memory address would be to broadcast breakpoints using the **asm-break** command. For example, on a stack-based architecture, if breakpoints were broadcast to all "STORE" machine instructions, then by having the command block for that broadcast set check the contents of the watched memory location, we could guarantee the ability to catch the inferior in the act of modifying that memory cell.

Therefore, the strongest claim we can make about TOPDOG's performance is that it will achieve a lower bound on the *expected* limits of possible performance, so long as the (dynamic) number of "STORE" machine instructions grows at least linearly with the number of instructions executed by an inferior program (usually a reasonable assumption). TOPDOG cannot be guaranteed to attain such a lower bound for every specific watch loop, because execution of the "STORE" machine instructions may not be distributed uniformly along the inferior's (dynamic) timeline. Instead, in a worst case scenario, a user might implement a watch by single-stepping over a segment of the inferior's timeline during which *no* "STORE" instructions at all were executed.

6.8. The Expressive Power of Loosely-Bound Watch Loops

We have devoted quite a lot of discussion — both analysis and synthesis — to loosely-bound watch loops or their equivalent, the **begin-watch** command. We do not wish to leave the reader with the impression that our concern was merely one of performance. Although realtime efficiency is undeniably an important consideration, it is not the only significant issue involved. Rather, there is a second semantic issue involved — no longer of integrity, but of power and expressiveness. So far, we have viewed loosely-bound watches as vague entities — one could gain a conceptual handle on them by their endpoints, which at least have seemed solid enough. For example, we have assumed that loosely-bound watches begin life at a breakpoint anchored to a fixed location in the inferior. But even in such cases, their termination conditions are exactly that — *arbitrary* conditions which may be a complex function of the data available in the inferior, the debugger convenience variables, and the event history database. Given the **wbreak** command (or its functional equivalent in these cases, the **$terminate_watch**() predefined function), we can easily compute far more involved termination conditions than merely that the inferior has exited from a particular frame. For that matter, the *initiation* of a loosely-bound watch could itself be "anchored" not solidly in a breakpoint, nor even in a nicely-behaved, comprehensible free-floating loop, but rather in another fuzzy, amorphous loosely-bound watch!

Should the user be encouraged (or even allowed) to express "events" such as "during the third interval in which *x* was greater than *y*, *z* took on only five distinct values"? Clearly, it would be possible, although not easy, to code this particular event "relatively efficiently", i.e., to embed its recognition code directly in a freestanding loop, so that an event was never raised in vain. Far more difficult would be subsequently to try to add code to recognize a second, similarly convoluted event within that same freestanding loop. The potential for logic errors in the recognition code might cause more false negatives (or even false positives) than might occur if two (or more) loosely-bound watches were used in an attempt to specify the two events "independently".

Fortunately, Dalek's dataflow approach to hierarchical events allows the user easily to write independent code to recognize such semantics. But it would still be costly to *efficiently* monitor the inferior for the semantics of the last example, unless the underlying hardware supported at least three watchpoint registers.

We wish to point out something the reader may not have noticed regarding what can be achieved through the expressive power of watch loops, or through their equivalent, our proposal for an augmented "when" construct, implemented as the **begin-watch** command in Figure 6-10. They permit a *qualitative* change in what can be expressed. In particular, they tremendously enlarge the universe of what can be expressed *declaratively*, without requiring detailed, *procedural* knowledge on the part of the user regarding a host of breakpoints that otherwise would need to be set at specific places to monitor for the desired conditions.

The user does not *need* to know exactly which places in the inferior's code might, e.g., alter *x*; rather, he only needs to know how to code a **begin-watch** command to monitor that variable. As he gains additional knowledge about the inferior, perhaps he realizes that he can speed up his watch considerably by making it only loosely-bound, because he is confident that the value of *x* is *never* changed during certain, well-defined intervals on the inferior's logical timeline. Alternatively, he may decide that *x* is *only* changed during certain intervals on the inferior's logical timeline. Note that the user now has the power to define the endpoints of any interval *declaratively*. In effect, that interval may itself be "floating", anchored only, but exactly, by the conditions defining another loosely-bound watch.

This expressive power is wonderful, but what will it cost in terms of efficiency? As we mentioned earlier, since most architectures do not support any hardware "watchpoint" registers, even a single free-floating event becomes very expensive. Yet consider what could be accomplished on an architecture that provided even two watchpoint registers in hardware. We would be able to watch, e.g., *x* and *y*, virtually for free throughout the entire course of the inferior's execution. That would allow us, at zero cost, declaratively to decompose the logical timeline into a host of (possibly overlapping) intervals, e.g., intervals in which ($x > y$), or ($x * y <= 1024$), or (*x* has remained unchanged for at least 500 milliseconds of the inferior's CPU time). If it were discovered that alterations to *x* and *y* — or certain properties of the intervals mentioned earlier — were reliable predictors of impending changes to other data structures, then several loosely-bound watches could be declaratively "piggybacked" onto the information gained from the efficient, hardware-assisted watches. Such opportunities for optimization should not be overlooked.

In conjunction with our programmatic approach to event recognition and programmatic access to the event history, the capability this provides for *reasoning* about intervals of the inferior's behavior should be exploited in future debugging methodologies. Perhaps in the near future, language-level support for *interval logics* [Hart85] [Schw83] will become standard equipment on debuggers.

6.9. Summary of Code Interference Problems and Solutions

In this chapter, we have delved into some rather dense subject matter and have amply demonstrated the complexity that may arise as a result of Dalek's full programmability. The semantics of re-entrant coroutines that may accidentally (or intentionally) activate one another by "tunneling" through the inferior are not to be taken lightly. Perhaps at this stage, the reader can more fully appreciate the configuration of control represented by the meta-backtrace display of Figure 4-6, and the benefits of the **dtrace**, **pause**, and **set exit** commands.

Since the fundamental requirement for any recorded event history is that it be *accurate*, false negatives and false positives alike are unacceptable. Fortunately, the coding convention we introduced in Figure 6-5 will prevent most false positives, and the coding convention shown in Figure 6-3 will prevent all false negatives caused by collisions between free-floating and bound primitive events. Since neither coding convention places an undue burden on the user, we can consider these particular problems closed.

Should the user desire to employ loosely-bound watch loops for more efficient realtime performance, such loops may "encounter" one another through a breakpoint in the inferior. Some such encounters are benign, but others will compromise the semantic integrity of the debugging computation. Such collisions can be managed through the interrupt priority model, but it is far better to avoid them altogether by employing the TOPDOG construction, and its associated new auxiliary commands and predefined functions. We have shown that — on the average and with certain reasonable caveats — then given any particular debugging objective, TOPDOG can achieve a lower bound on the number of strides required to reach that goal and collect the desired observations along the way.