CHAPTER 4

User's View of Dalek: Meta-Debugging and More Predefined Functions

In this chapter, we conclude the presentation of Dalek as it appears to the user. Several of the topics that appear here might have been considered earlier, but either because they deal with more advanced features or involve more detailed examples, the discussion was deferred to the current chapter. We first describe Dalek's discipline of stacking partially-interpreted command blocks and enumerate some of the circumstances in which this ability comes into play. The degree of complexity that is capable of developing under this regime leads rather naturally into a discussion of meta-debugging. Later sections cover certain interesting predefined functions and give some insight into the debugging methodology supported and encouraged by Dalek. We close with an example illustrating Dalek code to measure the conditional probability that a push onto the inferior's runtime stack will be followed by another push, rather than by a pop. As with the previous two chapters, the reader is encouraged to refer periodically to the Appendix, with its quick reference guide to Dalek's syntax. The index at the beginning of the Appendix also includes Dalek's predefined debugger functions, which feature prominently in this chapter.

4.1. Stacking Dalek Command Blocks

The fact that, in certain situations, Dalek can stack a command block that it has only partially interpreted in order to begin interpretation of a new command block should come as no surprise to the reader. After all, without this capability, user-defined functions could only be called directly from Dalek's interactive command level. Yet as shown in Figure 3-8, **$deleted_during_interval**() is repeatedly called from within a **while** loop. Indeed, if it were impossible to call a user-defined function — e.g., within the command block of a breakpoint or of an event — without "losing" the remainder of the commands in the calling block, that would force the user repeatedly to write the code for the function "in-line", thus defeating the whole purpose of user-defined functions.

In Dalek, unlike gdb, partially-interpreted command blocks will *always* be stacked, pending their resumption, when the focus of interpretation shifts to a different command block. This practice is obviously the natural way to handle calls to user-defined functions. It is also clearly beneficial in that, after a bound primitive event has been raised at a breakpoint and Dalek has finished interpreting the command blocks of any high-level events that may have been triggered, Dalek is able subsequently to resume its interpretation of the remainder of that breakpoint's command block. Similarly, after activity in the dataflow graph initiated by the raising of a free-floating primitive event has ceased, Dalek is able to resume interpretation of an underlying **while—step** loop in which the **event raise** command for that free-floating primitive event may have been embedded.

What may not be so readily apparent are the ramifications of this practice when a transfer of control between different command blocks is not direct, as with a call to a user-defined function or the raising of an event, but is instead effected by "tunneling" through the inferior process — that is, when the transfer of control is mediated by the inferior process!

As a concrete example, suppose Dalek is interpreting command block *A* when it encounters a **step** command. It will first instruct the inferior process to execute the object code for the next line of source code, then wait for the inferior to report back. But if the inferior encounters a breakpoint at the end of that step, Dalek will stack the remaining commands in block *A*, so it can begin to interpret this new breakpoint's command block, *B*. If Dalek's interpretation "falls off the tail" of block *B* (i.e., the last command in *B* does *not* transfer control back to the inferior) then Dalek will immediately pop its command stack and resume interpretation where it left off in block *A*,[1] beginning with the command following the original **step**. Even if the last command in *B* were, e.g., to **cont**, thus ceding control to the inferior, Dalek would eventually resume its interpretation of block *A* once it regained control from the inferior, although this might not occur until after the inferior's termination! This practice stands in sharp contrast to the policy of other debuggers (including dbx, gdb, and Saber-C), which discard all commands in a block following a **step** or **cont**.

Much more could (and will) be said regarding the semantics of this subject, but for the moment, we are content to offer an example demonstrating the need for this ability to stack command blocks, *especially* in those cases where a transfer of control between command blocks is mediated by the inferior process. Obviously if we wish to employ free-floating loops to watch inferior variables, we need such a capability. The fact that the flow of control "tunnels" through the inferior and re-emerges in the same command block is just a fortuitous special case. In the following section, we present an entirely different such special case where the ability to resume interpretation of a command block after control tunnels through the inferior is also required.

---

[1] The semantics of "falling off a command block's tail" are still entirely consistent with those of gdb, and encompass them as a special case. Both debuggers resume interpretation in the previous "block", but for gdb, that will *always* be the top interactive command level.

Further treatment of this topic appears in Sections 5.4.3 and 5.4.7, which focus primarily on implementation issues, although important semantic matters are centrally involved there as well. The entirety of Chapter 6 is devoted to a class of semantic issues specifically raised by Dalek's ability to stack command blocks that are activated by transfers of control that have "tunneled" through the inferior.

4.1.1. Example — How Useful *are* Embedded 'step' Commands?

Dalek enables the user to practice debugging methods other than those typically employed within tools such as gdb and dbx. For example, commands such as **stepi**, to single-step one machine instruction, may appear part way through a block of breakpoint commands. Unlike in gdb or those versions of dbx that support breakpoint commands, commands following **stepi** in a Dalek command block *will* be interpreted. Although such constructions are not typically necessary, in certain situations their availability allows users to solve otherwise intractable problems. A case in point is the following real-life situation, over which the author personally stumbled.

Suppose the user is interested in the number of times function *foo* is called. Or for purposes of optimization, he may want to know which procedures call *foo* most frequently, or collect statistics regarding the values typically passed to *foo* as incoming parameters. Therefore the user instruments the entry to function *foo* with a breakpoint in order to raise an event capturing these essentials of *foo*'s invocation. The commands in Dalek would look something like those shown in Figure 4-1. For the majority of functions, this command block will adequately perform its intended purpose. Yet suppose the C language source code for *foo* begins as shown in Figure 4-2.

```
break foo
commands
   event raise 'foo ( $func(1), parm1, parm2 )
   cont
end
```

Fig. 4-1 The user intends to raise event ‘foo whenever *foo* is called.

```
int foo (parm1, parm2)
   register int parm1, parm2;
{
   while (parm1 > parm2++)
   .
   .
   .
```

Fig. 4-2 C language code for beginning of procedure *foo*.

Now gdb (and also Dalek) will interpret "**break** *foo*" to mean set a breakpoint at the beginning of the *first executable source statement* in procedure *foo*. Unfortunately, doing so will cause event ‘foo to be raised *at every pass* through *foo*’s "while" loop, because the first machine instructions generated by most compilers for that statement will be the code to evaluate whether (*parm1* > *parm2*++). The user would have unknowingly set a breakpoint at a location that was the target of a branch instruction. Needless to say, the spurious events raised at this breakpoint would give extremely misleading information to the user!

In general, allowing the user to issue a command like "**break** *foo*" without forcing him to actually look at the source code for *foo* should be considered a beneficial feature, in Dalek as well as in gdb. Yet clearly, in some circumstances, this feature can be problematic. If we do not wish to prohibit such a friendly syntactic feature altogether, is there perhaps some alternative semantic interpretation that is less problematic? Suppose that, instead of setting the breakpoint at the beginning of the first executable source statement in *foo*, Dalek (or gdb) were to set it at the first *machine instruction* in *foo*. But if this were done, the breakpoint commands above might not work as desired. This is because on many architectures (e.g., Sun's 680x0 or SPARC workstations) some "fuzz" is required as a prologue to every function to adjust the frame-pointer register, and sometimes (as in this case) to copy incoming parameters off the stack into registers.

The ideal solution is to locate the breakpoint at *foo*'s first machine instruction, but then to single-step several machine instructions so that the current frame is properly established before raising event ‘foo. Thus, for a Sun3, the commands in Dalek should look like those of Figure 4-3.

It might seem that the user, having gone to all the trouble of inspecting the machine instructions in *foo*'s prologue so that he knows how many **stepi** commands to use in Figure 4-3, might as well just set the breakpoint on the last such instruction and dispense with *any* **stepi** commands. Yet (unless the compiler issued some NOP instructions) we must assume that *every* instruction in the prologue is essential to properly establishing the frame, loading the register parameters with values from the stack, etc. By raising an event and recording information *before* the situation has stabilized, he would risk record-

```
break  &foo     # Break at first machine instruction in foo.
commands
   stepi        # Take enough steps to cross the prologue code.
   stepi
   stepi
   event raise  `foo ( $func(1), parm1, parm2 )
   cont
end
```

Fig. 4-3 Ensuring a one-to-one correspondence between *foo*() and ‘foo.

ing misleading data.[2] Alternatively, by setting the breakpoint anywhere beyond *foo*’s prologue code, a spurious instance of the event would be raised on every pass through *foo*’s "while" loop.

Here then, is a bona-fide debugging situation that requires the ability to interpret debugging commands following a **stepi** in a command block.

4.2. Meta-debugging

Previous sections have contained some fairly extensive examples of code written in Dalek’s debugger language. Code such as that in Figure 3-7 for the user-defined debugger function **$deleted_during_interval**() or that in Figure 3-6 for the high-level

[2] At one point, some implementation effort was devoted toward getting Dalek to recognize when the inferior was in such an "unstable" situation so it could still return semantically correct answers when asked, e.g., to print the value of a register parameter that had not yet been moved off the stack into its designated register. That work was put in abeyance because details of the solution(s) are extraordinarily dependent both on machine architectures, and also on poorly-documented conventions used by various compilers in their symbol table formats.

event ‘match illustrate a degree of complexity that is routinely achieved in practice, even during relatively nonstrenuous debugging and program exploration activities.

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

One such introspection facility is verbose mode, which can be selectively enabled by the user. In this mode, every command, along with the name and type of its enclosing block, is displayed as it is interpreted. The astute reader will recognize here a debugger-level analog of that feature, common to many debuggers, for *tracing* lines in the inferior’s source code as they are executed. Merely making a programmable debugger’s actions visible allows many simple errors to be found easily.

We also provided several means to pause Dalek during its interpretation of the user’s debugging program. This permits the user to examine the state of the inferior process, and, equally importantly, that of the debugging program itself. To remind the user of its paused status, Dalek alters its prompt. For example, if the prompt was originally "<dalek>", and Dalek pauses inside a user-defined function named **$glarb**(), then the new prompt will be

"<dalek> (PAUSED) [FUNC$glarb]> ".

By thus pausing, it becomes possible to alter data or control flow in the debugging program. For example, if the debugging program is paused inside an event, values of that event's attributes can be manually inspected and altered. The user might elect to override the debugging program's logic by breaking out of an enclosing **while** loop, or could force a user-defined debugging function to prematurely return a specified value. In addition, the user can **edit** any block of code in the debugging program (including those currently stacked) before issuing the **unpause** command and allowing Dalek to resume interpretation of the underlying command blocks.

The user can embed a **pause** command within a block of debugger commands, or can achieve the same effect by pressing the interrupt key (e.g., Control-C), although with the latter method it is obviously difficult to pause at a particular line of debugging code. To accomplish that purpose, the user can set a meta-breakpoint at any command-line in his Dalek language program by **edit**ing the relevant command block.

Dalek's **edit** facility is primitive, but adequate. Although it certainly makes Dalek more user-friendly, we would not have bothered to implement **edit** were it not for the fact that its general capabilities are required to support the setting and clearing of meta-breakpoints, an interesting research topic we wished to pursue. When **edit**ing a command block, one line is displayed at a time, and the user can issue single-letter commands. For example, **L** will list the entire command block before repositioning the cursor on the currently displayed command-line; **D** will delete the displayed command-line, splicing any adjacent command-lines together; **E** will allow the user to edit (in overwrite mode) the displayed command-line; **P** will select the previous command-line; and **B** will set a meta-breakpoint at the displayed command-line.

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

The **dstep** command takes a numeric parameter — if it is negative, then **dstep** mode will be engaged immediately and indefinitely; if zero, then **dstep** mode will be cancelled; if positive, then **dstep** mode will be engaged, but not until *after* Dalek has interpreted that number of embedded command-lines.

Specifying a positive parameter for **dstep** is akin to setting an alarm. This allows the user to check on Dalek's progress in the future even though he may not know exactly which breakpoints the inferior will hit, which events will be triggered, or which user-defined debugger functions will be called. In such situations, the user would not necessarily know *where* to set meta-breakpoints. This deferred **dstep** mode can also be helpful if, e.g., the user wants Dalek to pause every hundredth pass through a **while—step** loop, but neglected to specify that when the loop was first written and invoked.

In addition to meta-tracing, meta-breakpoints, and meta-single-stepping, Dalek also supports a meta-backtrace. Following the convention pioneered by verbose mode, the meta-backtrace display shows the type and name of a command block along with the next command pending in that block; but it does this for *every* stacked command block, not just for the block Dalek is currently interpreting. When Dalek is paused, this feature allows the user to get a complete picture of its underlying status.

Figures 4-4 through 4-6 illustrate how **dtrace**, Dalek's meta-backtrace command, can help to clarify even extremely convoluted debugging situations.

The story begins innocently enough at Dalek's TOP_LEVEL. The user sets meta-breakpoints at three different command lines associated with inferior breakpoints #1, #2, and #3 respectively, and then gives the **run** command to start the inferior. The inferior executes breakpoint #1, and it is then suspended while Dalek interprets the commands for breakpoint #1. When Dalek encounters the meta-breakpoint in that command block, it pauses and stacks the commands still pending in that block before reopening an interactive dialogue with the user. At this point, the meta-backtrace display appears relatively benign, as shown in Figure 4-4. The numbering convention at the left of that display corresponds to that used by gdb (and Dalek) in the **backtrace** display for the inferior process's runtime stack. As shown in Figure 4-4, the display encloses the next pending command of a block in angle brackets to make it visually more distinct from the name of the parent block itself, which is enclosed in square brackets. The suffix "<BP>", appearing on the same line after a pending command, indicates that a meta-breakpoint is set on that command-line.

Before giving the **unpause** command and allowing the inferior to resume its normal exe-

```
<dalek>  (PAUSED)  [BP#1]> dtrace
   0    [PAUSE]
   1    [BP#1]
        < cont > <BP>
   2    [TOP_LEVEL]
```

Fig. 4-4 A relatively benign meta-backtrace display.

cution, the user decides to check whether one of the inferior's data structures contains a cycle. So while Dalek is paused, he invokes one of the inferior's functions, *has_cycle*(). Rather than doing that directly (i.e., "**print** *has_cycle*()"), he follows a more circuitous route and embeds that command within a new user-defined command (UDC) he names **cycle-check**. He then issues this new command, but it happens that one of the inferior breakpoints (#2) he had previously set is located in the inferior function that **cycle-check** has just called. Not only that, but he had also set a meta-breakpoint at one of the commands associated with that breakpoint. When Dalek encounters that meta-breakpoint, it dutifully pauses in its interpretation of the commands for breakpoint #2, and once again initiates a new level of dialogue with the user. The meta-backtrace for this situation appears in Figure 4-5. As shown, Dalek's prompt includes an extra "(P)", indicating that multiple **pause** levels underlie its current state.

Undaunted by the intimidating status shown in Figure 4-5, the user proceeds interactively to write and then **call** a new user-defined function (UDF), **$will_finish**(). Among other commands, this contains one ordering the inferior to **finish** executing its current procedure. But this is not the user's lucky day: when the inferior attempts to finish its

<dalek> (PAUSED) (P) [BP#2]> **dtrace**
0   [PAUSE]
1   [BP#2]
< **echo** Ending breakpoint #2 now.\n > <BP>
2   [STACK_DUMMY *has_cycle*]
3   [UDC cycle-check]
< **echo** End of UDC that calls the inferior function, "has_cycle".\n >
4   [PAUSE]
5   [BP#1]
< **cont** > <BP>
6   [TOP_LEVEL]

Fig. 4-5 The meta-backtrace display reflects increasing complexity.

current procedure, it encounters yet another breakpoint, whereupon control transfers back to Dalek once again. Dalek begins to interpret the command block for breakpoint #3, but it again encounters a meta-breakpoint and is forced to pause a third time. Figure 4-6 shows the meta-backtrace display at this stage. It is not a pretty sight. Dalek’s prompt now includes two extra "(P)" strings in addition to the normal "(PAUSED)" reminder, reflecting the fact that *three* different read-eval-print loops are currently stacked. At this stage, the user would be well advised to issue a command to "**set exit** 8", instructing Dalek to immediately terminate the 8 most recent pending command blocks. In response, Dalek would pop the command blocks labelled 0 through 7 in Figure 4-6, and would continue activity in the block labelled 8 in that same display. Thus the user’s interactive dialogue with an earlier **pause** would be resumed. Should the user then give another **dtrace** command, he would be relieved to see the uncluttered display shown in Figure 4-4, the first in this series.

```
<dalek>  (PAUSED) (P) (P) [BP#3]> dtrace
   0    [PAUSE]
   1    [BP#3]
        < echo Starting breakpoint #3 now.\n > <BP>
   2    [FINISH has_cycle]
   3    [FUNC$will_finish]
        < echo End of UDF $will_finish().\n >
   4    [PAUSE]
   5    [BP#2]
        < echo Ending breakpoint #2 now.\n > <BP>
   6    [STACK_DUMMY has_cycle]
   7    [UDC cycle-check]
        < echo End of UDC that calls the inferior function, "has_cycle".\n >
   8    [PAUSE]
   9    [BP#1]
        < cont > <BP>
   10   [TOP_LEVEL]
```

Fig. 4-6 Another fine mess the user has gotten himself into!

4.3. Example — Making Permanent Copies of Inferior Strings

Dalek currently has several families of predefined debugger functions. Some support event operations and access to the event history database, while others provide timing capabilities or allow various operations related to the inferior’s runtime stack. Still other debugger functions do not fit neatly into any particular category, but nevertheless are undeniably useful. One such miscellaneous predefined debugger function is **$savestr**(). To best describe its purpose, we must first digress slightly.

The *char ** type is ubiquitous in C programs; it is preferable to the alternative (an *array* of *char*) in a variety of situations. One drawback to debugging a program that makes extensive use of *char* * is that often the user has no way to take a snapshot of the current contents of such strings. Even in debuggers, such as gdb, that support convenience variables, issuing a command to:

set $string = (char *) *inferior_string*

will not make a copy of the current *contents* of the string itself, but rather a copy of the *address* of that string. Although this is exactly what is desired in some situations (e.g., the ‘addr attributes of the ‘malloc and ‘free events in Figure 3-6), it can be very annoying at other times. Often, a variable like *inferior_string* may hold the same address for quite a while, yet the contents of the string at that location may undergo frequent changes in composition and length. Issuing a command to:

set $string_contents = * (char *) *inferior_string*

will not solve the problem, because it will merely make a copy of the *single character* at the beginning of the string.

Thus it was necessary to invent a function like **$savestr**(). The command:

set $string_contents = **$savestr** (*inferior_string*)

will indeed save a copy (in Dalek’s address space) of the current contents of that string. This is useful whenever a permanent record is desired, for example, as the value of an event’s attribute. In another typical application, the user may wish the inferior to execute until the contents of a string are altered. This is easily accomplished by the following commands:

```
set  $string_contents =  $savestr (inferior_string)
while ( $string_contents == inferior_string )
   step
endwhile
```

4.4. Examples — Learning about Callers via Predefined Debugger Functions

In many circumstances, it can be advantageous to programmatically glean information about functions whose activation records are currently stacked on the inferior process's dynamic call chain.

Perhaps the simplest such example involves the desire to reestablish interactive control at a breakpoint in procedure *foo*, but only when it is directly called by procedure *goo*. This can be easily accomplished by the code shown in Figure 4-7.

A real-world situation that is considerably more convoluted involves the memory allocation example shown earlier in Figure 3-6. A suggested enhancement was to define an

---

```
break foo
commands
   silent  # Suppress default announcement of encountering a breakpoint.
   if  ( "goo" == $func(1) )
      pause                  # Reestablish interactive mode.
   endif
   cont                      # Continue execution of the inferior.
end
```

Fig. 4-7 Dalek code to break whenever procedure *foo* is called by *goo*.

---

additional attribute in ‘free to hold the name of its immediate caller, determined by invoking **$func**(1). With this information available, if the inferior process ever attempted to *free()* an address that had already been allocated and freed earlier, it would be easy to find the places in the source code that were responsible for causing that bug.

But on certain systems (e.g., Sun3 running UNIX 4.2 Release 3.4), the plot is much thicker than it might at first appear. On such systems, it can happen that *free()* is indirectly called by *malloc()* itself! For example, this always happens the first time any inferior calls *malloc()*. The calling sequence is as follows:

malloc ()
_malloc_at_addr ()
free ()

Several library or system calls, e.g., *exit()*, *initscr()* and *wrefresh()*, invoke *malloc()* or *free()*. Occasionally the memory blocks thus allocated and de-allocated cannot be matched with each other, probably because of intervening calls to *realloc()*. Although we could define a third primitive event, ‘realloc, and include it among the constituents of ‘match, the code for ‘match would need to be more complex. Besides, if the user’s source code never calls *realloc*() directly, it seems unfair that he should have to write debugging code for it.

In either case, if the user deems it relatively unlikely that calls to memory allocation routines by library routines will exhibit bugs, then he may wish to avoid recognizing *any* primitive allocation event when *malloc()* or *free()* is invoked on behalf of a library routine — especially because of the extra time required for event ‘match to search its needlessly lengthy queue of ‘malloc tokens every time it is triggered by the arrival of a token from ‘free. One way to distinguish such library-induced calls would be to write debugger code in the gory, ad hoc style of Figure 4-8.

A far better method would be possible if there were some way programmatically to determine whether or not a given inferior function was compiled with a "-g" flag. The predefined debugger function **$g_frame**() was provided for just such a purpose. It will return the number of frames above the currently selected frame where the nearest calling procedure that was compiled with the "-g" option is located. Because standard libraries are invariably compiled *without* this flag to avoid storing "unnecessary" debugging information, **$g_frame**() will, in general, allow us programmatically to distinguish between library routines and other routines currently being debugged by the user. Thus Figure 4-9 shows a much cleaner way to raise an event in *malloc()*, on *only* those occasions when it

---

```
break malloc
commands
   silent
   set $caller = $func(1)
   if ( $caller != "_malloc_at_addr"  &&  \
       $caller != "_set_subwin_"      &&  \
             etc.                     &&  \
             etc.                     &&  \
       $caller != "newwin"                 )
      event raise ‘malloc
   endif
   cont
end
```

Fig. 4-8 Ad hoc Dalek code to distinguish system-library-induced calls to *malloc*.

---

is directly invoked by a user-level routine.

Alternatively, the user may only want to capture ‘malloc events that were invoked (either directly or through library routines) from source code residing in a suspicious file. Figure 4-10 shows how two predefined debugger functions can be composed to accomplish this purpose. Note that this code is *not* designed to detect situations where a procedure in "buggy_file.c" calls a routine in a different file that *was* compiled with the "-g" flag, which, in turn, finally calls *malloc*(). If that is the user’s intention, he should structure the code with a loop, similar to that shown in Figure 4-11.

Another "frame-walking" predefined function is **$t_frame**(). It will return the number of frames above the currently selected frame where the top-level frame is executing. This can be useful, for example, when the user wants to programmatically determine the maximum depth reached by the inferior process’s runtime stack. The application shown in Figure 4-11 utilizes **$t_frame**() to track which procedures are *even indirectly* called by

```
break malloc
commands
   silent
   if ( 1 == $g_frame() )
   # Is our caller the nearest to have been compiled with the "-g" flag?
         event raise ‘malloc
   endif
   cont
end
```

Fig. 4-9 A preferred alternative to the code of Figure 4-8.

```
break malloc
commands
   silent
   if ( "buggy_file.c" == $file ( $g_frame() ) )
      event raise ‘malloc
   endif
   cont
end
```

Fig. 4-10 Which invocations of *malloc* arise from a particular file?

an inferior procedure named *foo*().

```
# This will print names of all functions even indirectly invoked by
# foo(). These could include library routines, if the user desires.

proc-break *
   silent
   set $top = 1 + $t_frame()
   while ( −− $top )
     if ( "foo" == $func ($top) )
        printf "Procedure %s was invoked on behalf of foo\n" $func (0)
        wbreak
     endif
   endwhile
   cont
end
```

Fig. 4-11 Print names of procedures even indirectly invoked by *foo*().

4.5. Example — Measuring Conditional Probabilities for Runtime Stack Behavior

Finally, having dispensed with several smaller, partly "expository" examples, we close with a more realistic and significant application for a debugger. Although we have previously used Dalek to gather timing and profiling information as well as to measure the maximum depth reached by the inferior's runtime stack, a more interesting application is to measure the *conditional probability* that a push onto the inferior's runtime stack is followed by another push, rather than by a pop. Typically, such information is quite relevant for CPU architects employing register windows. In addition, if an inferior were to demonstrate a high conditional probability that a pop follows another pop, that behavior would be consistent with a high degree of tail recursion. If tail recursion were indeed the cause, then whether or not it presented a performance or safety problem would depend on the inferior and on the underlying architecture.

The code in Figures 4-12 and 4-13 uses a set of breakpoints broadcast to all procedures. Instead, we could have accomplished this measurement by means of a **while — stepi** loop similar to the variant of Figure 3-2 that was discussed in Section 3.2.2.1. However, that method would perform much slower than the preferred method of using broadcast breakpoints due to the overhead incurred by all the calls to *ptrace*() when single-stepping.

We could also have used a different broadcast mode — the **asm-break** command — to set breakpoints at every "procedure call" and "return" machine instruction. In that case, the breakpoint code of Figure 4-12 would be divided between the two new broadcast sets. We would dispense with the call to **$quiet_finish**() and assign the code following that

```
proc-break *
   silent
   if ( "_exit"  == $func (0) )
      set $running = 0
      # End of computation, since "_exit" neither calls nor returns.

   else if ( 0 == $t_frame() )
      # Do nothing.  We are running in the top frame; there is no caller,
      # so no "push" to count.  Certain procedures run on an empty stack,
      # e.g.,  start, start_float, finitfp_, main, and exit.
      # Since Dalek implements $quiet_finish() by setting an "up-level"
      # breakpoint in the calling frame of the current procedure,
      # we cannot $quiet_finish() in such procedures.
      # Instead, we allow control to return to an outer while — cont loop.
   else

      # We just "pushed" into a new stack frame so update our running totals:
      if ($last == "push")
         set $push_push++
      else if ($last == "pop")
         set $pop_push++
      endif
      set $last = "push"

      call $quiet_finish()
      # When the call to $quiet_finish() completes, we will have
      # just "popped" out of an old stack frame so update our running totals:
      if ($last == "pop")
         set $pop_pop++
      else if ($last == "push")
         set $push_pop++
      endif
      set $last = "pop"
   endif
   # Let execution "fall off our tail" into a stacked Dalek command block.
end
```

Fig. 4-12 Breakpoint code to measure conditional probabilities for stack growth.

```
# Perform some initialization chores:
set $last = "none"
set $push_push = 0
set $push_pop = 0
set $pop_pop = 0
set $pop_push = 0
set $running = 1

run

# After each "outermost" call to $quiet_finish() has completed,
# there are no more stacked Dalek command blocks to propel the inferior
# so we need the following loop to move execution along:
while  ($running)
   cont
endwhile

# Print out the results:
set  $total_pushes = $push_pop + $push_push
set  $total_pops = $pop_pop + $pop_push     # (Yes, these are equal.)

printf "Conditional probability that a PUSH is followed by a PUSH == %f\n", \
                         (float) $push_push / $total_pushes
printf "Conditional probability that a PUSH is followed by a POP == %f\n", \
                         (float) $push_pop / $total_pushes
printf "Conditional probability that a POP is followed by a POP == %f\n", \
                         (float) $pop_pop / $total_pops
printf "Conditional probability that a POP is followed by a PUSH == %f\n", \
                         (float) $pop_push / $total_pops
```

Fig. 4-13 ‘Outer’ code to measure conditional probabilities for stack behavior.

call in the original broadcast set to that for the "return" machine instructions. Note that this method (using the **asm-break** command) would work properly even for an inferior whose executable file had been "stripped" of all debugging and symbolic information.