CHAPTER 2

A User's View of Dalek: Introduction

This chapter begins a description of Dalek "as it appears to the user". It focuses on just a few of Dalek's basic features. Events are covered in Chapter 3, and Chapter 4 rounds out the user's view by exhibiting some of Dalek's more advanced capabilities. Since Dalek's benefits do not depend on a graphical interface, we can thankfully dispense with scads of figures illustrating various window configurations. Nevertheless, the task of these three chapters is still somehow to convey the "look and feel" of Dalek.

We assay our goal via a series of examples, interspersed with discussion, to illustrate the versatility of Dalek's toolset in a wide range of situations. Although a few of the examples are admittedly somewhat contrived (to facilitate the exposition), more realistic examples are presented as well. The reader may find it helpful periodically to refer to the Appendix, which contains a quick reference guide to Dalek's syntax. An index to Dalek's commands and also to its predefined debugger functions appears at the beginning of the Appendix.

2.1. Notation and Syntactic Conventions

For the reader to easily understand Dalek language code, it will help to keep the following syntactic conventions in mind:

- Names of convenience variables and Dalek's predefined and user-defined functions start with a '$'.

- Names of events and their attributes start with a backquote, e.g., ‘foo.
- Comments begin with a ‘#’ and continue to the end of the line.
- A ‘\’ indicates that the current line is continued on the next.
- Dalek commands and predefined $functions appear in boldface font.
- Names of objects in the inferior are italicized.

2.2. Excerpts from Sample Dalek Sessions

Many of the following examples make no mention of bugs. Although it could be argued that this tack is misleading since Dalek is supposed to serve as a *debugger*, we feel this approach is justified, partly because Dalek’s other purposes include serving as a tool for "program exploration", maintenance, and testing. In addition, quite often even though the user’s ultimate goal may be to discover and isolate the causes of a bug, his initial maneuvers are instead designed to familiarize him with certain aspects of the program’s dynamic behavior. We liken this process to that of a software engineer practicing top-down coding — one does not proceed to the esoteric or nitty-gritty aspects of a program until one has first gained some semblance of mastery over the basic control flow.

2.2.1. Example — Exploring an Unfamiliar Program

Suppose the user must debug a large program with which he is unfamiliar. Perhaps the program was written by others; or possibly the user was involved in its writing, but the experience was so traumatic that most memories of its structural details and of the relevant design decisions have mercifully been blotted from his mind. A useful first step might be to issue a command inherited from gdb:

info function <regular-expression>

In response, Dalek will display the names of all procedures in the inferior program whose names match the given regular expression.[1] Next the user might want to briefly scan the source code of one of the matching procedures. As in most debuggers this is easily accomplished. For example, in Dalek:

list *foo*

Another helpful feature inherited from gdb might be useful at this point: The user can search (forwards or backwards) from the currently displayed source code for a specified regular expression. Perhaps the user wants to find the closest line where procedure *foo* references variable *bar* by name:

search *bar*

So far none of the features we have used were concocted solely for Dalek, and they are probably not even unique to gdb. These features exemplify a certain degree of integration between a debugger and a (read-only) text editor. The same functionality could have been achieved (even on a single-window CRT) by interrupting the debugger and starting *vi*. Then the user could (build and) search a TAGS file[2] for interesting procedure names. Next he could view a selected procedure's code by editing the appropriate file.

The advantage to integrating these features into a debugger is simply that they are commonly occurring actions, and that making them available within the debugger allows the

---

[1] Unfortunately, Dalek (like gdb and *vi*) is limited in its use of regular expressions. Disjunction is not supported.

[2] The UNIX utility *ctags* will create a file that maps names of C functions to the names of the files in which those functions are defined.

user to avoid extra "mental context switching". In Dalek and gdb, if the inferior stops at a breakpoint, a **search** command will be construed to mean search forward from the line of source code that is currently executing. Even in a multiple-window environment, if there were no integration between the debugger and an editor, the user would still have to manually instruct the editor to position its cursor at the code for the appropriate procedure when a breakpoint was encountered. Thus, with some integration, the user is more likely to avail himself of these features, and is less likely, after switching from the debugger to an editor, to get stuck when he discovers that he has forgotten *why* he undertook his present course of action in the first place. By decreasing the mental and temporal cost of this functionality, we shift the break-even point and allow such actions to become cost-effective in a wider range of debugging situations.

Meanwhile, back at the debugging session, the user has finally advanced to a stage where some of Dalek's unique capabilities come into play. Having learned of the existence of a set of procedures whose names match a certain regular expression, the user now desires to get some idea of when those procedures are called during execution of the program. To accomplish this purpose, the user can provide a selective trace of the inferior's execution by "broadcasting" breakpoints to all those matching procedures, as shown in Figure 2-1. The **proc-break** command will cause the specified command block to be shared by every breakpoint in that broadcast set. Clearly it would not be very informative if that shared command block were to print out the same hardcoded message at every breakpoint in that set. Instead, this generic command block must somehow be customized. Dalek's predefined debugging function **$func**(N) will return (as a string) the name of the calling procedure N frames up in the inferior's runtime stack; in particular, **$func**(0) will

```
proc-break  <regular-expression>
   silent   # Suppress default announcement of hitting a breakpoint.
   printf  "Entering function %s\n",  $func(0)
   cont     # Continue the inferior process.
end
```

Fig. 2-1 Dalek code to broadcast breakpoints to selected procedures.

return the name of the currently executing procedure. This allows the shared command block of Figure 2-1 to print out a different, customized message at every breakpoint in the broadcast set.

Once again, it might be objected that such selective tracing could be accomplished under dbx as well. After interrupting dbx, the user would *egrep* through the TAGS file for <regular-expression>, redirecting all output into a file. Next he would edit that file: On each line, he would insert "trace" before the procedure's name, and would delete all text trailing after the procedure's name. After resuming the dbx session, he would **source** that file into dbx.

Yet this alternative to fully-programmable debugging has drawbacks similar to those of the alternative in the previous example: Such a sequence of actions is convoluted and error-prone. Even if the user "compiles" these actions into a shell script (which invokes *awk* or *sed* for the editing), the relatively high cost of the mental context switching would preclude the use of this technique in many situations. In addition, even slightly more complex breakpoint actions would be beyond the reach of dbx, since it lacks a rich enough language to do much of anything except for tracing. Gdb, without Dalek’s extensions, is similarly disadvantaged.

The example of Figure 2-1 was abstract in at least two senses — the intent in writing the debugging code was somewhat fuzzy (the user desired to "get some idea" of when certain procedures were called during execution) and the specific breakpoint commands were themselves completely generic.

2.2.2. Example — A Bug in *lex*-generated Code

We now feel compelled to present the obligatory example of a real-life bug that was actually encountered. This will hopefully satisfy the reader’s implicit question, "Can Dalek truly help in ferreting out *real* bugs?". Having thus eased the reader’s mind on this account, we will afterwards feel free to proceed to more interesting examples.

The bug in question was in C code that had been automatically generated by the UNIX *lex* utility, which generates C code to realize a table-driven finite automaton. Rather than practicing "debugging in 20/20 hindsight", we will show exactly how, in practice, we actually did isolate the cause of the bug. The whole exercise took less than ten minutes of elapsed time. This was in spite of the fact that the bug was found by a "committee" — two people cooperated to find the bug: one knew nothing about the application program and the other knew nothing about debugging.

When the *lex*-generated code was executed, the result was that a segmentation fault occurred within procedure *yylook*() during a reference to a table element. This outcome was the result of executing the C statement:

yyt = yystate−>yystoff;

When the segmentation fault occurred, it was easy to discover that its proximate cause was the de-referencing of an out-of-bounds pointer, *yystate*. But having worked backward from a blatant symptom to its proximate cause did not provide sufficient information to prescribe a cure in this case. To do that, we needed to find where *yystate* was first assigned its bad value.

Our next step was to set a breakpoint at procedure *yylook*() and reexecute the inferior, but our patience soon wore thin as we discovered that this procedure was invoked quite frequently. We therefore maintained a counter among that procedure's breakpoint commands to determine the particular invocation of *yylook*() that resulted in the segmentation fault. The command block to accomplish this, shown in Figure 2-2, would work fine under gdb, as it does not yet draw on any of Dalek's unique capabilities.

```
set  $count = 0
break  yylook
commands      # Specify a command block for the newest breakpoint.
   silent   # Suppress default announcement of hitting a breakpoint.
   set  $count++
   cont     # Continue the inferior process.
end
```

Fig. 2-2 Dalek code to count invocations of a given procedure.

Re-executing the inferior showed which invocation of *yylook*() caused the segmentation fault. Let's suppose $count was 43 when the fault occurred. We next modified the breakpoint code so it would continue execution of the inferior for all invocations of *yylook*() except the last, fatal one. This was easily accomplished by using Dalek's **if** statement,[3] as shown in Figure 2-3.

```
set  $count = 0            # Re-initialize our counter.
commands      # Re-define the command block for the newest breakpoint.
   silent         # Suppress default announcement of hitting a breakpoint.
   if  ( ++$count  >=  43 )
      pause                 # Resume interactive mode
      set exit 1
      # Exit this command block.  See Section 5.4.4 for why this coding
      # convention is preferable to the more straightforward way.
   endif
   cont                        # Continue the inferior process.
end
```

Fig. 2-3 Dalek code to pause once *yylook* is called 43 times.

Once we regained interactive control, we knew we were fairly "near" the segmentation fault. At this stage, we could afford to spend the time to single-step the inferior as long as *yystate* (the object whose de-referencing ultimately manifested the segmentation fault) held a value that pointed within the bounds of the *yysvec* table. At this point, we interactively issued the following Dalek commands:

[3] Some other debuggers can accomplish the same end through a completely different (and considerably more ad hoc) syntax. For example, in gdb a breakpoint can be "ignored" the first 42 times it is executed.

```
while ( yystate > yysvec && yystate – yysvec < \
                        sizeof (yysvec) / sizeof (struct yysvf) )
    step
endwhile
```

When Dalek exited from this **while** loop, the inferior was positioned at the first line at which the value of *yystate* went out of range:

**lsp++ = yystate = yyt–>advance + yysvec;*

In the above line of source code, variable *yysvec* is the base of the finite automaton’s state table, and *yyt* points to a data structure embodying the current state transition rule. This line of source code is intended to shift the finite state automaton into its next state; thus *yyt–>advance* is the offset of the new state from the beginning of the table.

The above line, then, is the real culprit. It sows the seeds of destruction that, much later, give rise to a segmentation fault.

From examining the inferior’s code, it was then easy to see that *lex* had specified a byte-sized (signed char) field for *yyt–>advance*. This was being used to index a large table, and the byte had overflowed.

The iterative technique used above to isolate a bug is not new by any means. What should be readily apparent, though, is that Dalek provides the right tools for assisting the user in this process.

2.2.3. Example — More Comprehensive Debugging Support

Several of Dalek's new features are neither astounding nor especially powerful, in and of themselves. Yet the impact of even one such incremental advance can produce a qualitative improvement in the debugging methods available to the user. One family of incremental advances involving breakpoints had an interesting evolution.

When Dalek originally diverged from gdb-2.6, it inherited one particularly undesirable "feature": the **commands** command required a parameter, namely the *number* of the breakpoint to which the commands in the following block (up until the **end** command) should be attached. Figure 2-4 is intended not only to refresh the reader's memory regarding gdb's syntax for breakpoints, but also to illustrate both debugger's responses to the **break** command. The crucial point to note is that the debuggers' responses contain the *number* of the breakpoint the user has just set. In gdb-2.6, it was mandatory that the user include this breakpoint number as a parameter to the **commands** command; without it, there was no way to attach a command block to a breakpoint.

(gdb) **break** *foo*
Breakpoint 5 at 0x20ae: file foo.c, line 23.
(gdb) **commands** 5 # The breakpoint's *number* is mandatory!
. . . # Specify the command block.
end
(gdb)

Fig. 2-4 Breakpoint command syntax in gdb-2.6 required a parameter.

Dalek's syntax does not force the user to specify a parameter for **commands**; rather, if no parameter has been specified, it defaults to the number of the last breakpoint that was set. This is an obvious way to make the debugger more friendly — so obvious, in fact, that the same default was later independently conceived by someone at the Free Software Foundation, who incorporated it into gdb by the time version 3.2 was released.

As users of Dalek, we had encountered several situations where it was desirable to set a breakpoint, attach a command block, and then **disable** that breakpoint for the time being.[4] Typically, a breakpoint would be temporarily disabled either for efficiency reasons, or because it would be unsafe to encounter it "prematurely" and interpret its commands. Often, the command to re-**enable** such a breakpoint would itself be embedded in the command block of a different breakpoint. Suppose, for example, there is a top-down parser with a particular bug that only manifests when parsing the first "if" statement *after* a "while" statement has been parsed. We might initially set a breakpoint in *parse_if*(), but immediately disable it. Then we would set another breakpoint in *parse_while*(), and have its command block contain a command to enable the breakpoint in *parse_if*().

Clearly, the syntax for any conceivable commands to enable or disable a breakpoint must allow a parameter so the user can indicate *which* breakpoint he means. Now in gdb, the *syntactic form* of this parameter must be numeric; it corresponds to the number contained in gdb's response message when that breakpoint was first set. We discovered that this requirement made debugging quite tedious. Often the commands to **break**, **enable**, etc., would be **source**d in from one or more files. It was therefore incumbent on the user,

[4] Disabling a breakpoint means that its "break" opcode will be removed from the inferior's code segment. However, the debugger still remembers that breakpoint's former location and command block so that the user can easily **enable** it again in the future.

when writing such **source** files, to keep mental track of which breakpoint numbers gdb would assign to which breakpoints. Another consequence of this restrictive policy was that it became effectively impossible for the user to **source** two files into the debugger, each of which set a breakpoint, disabled it, and later re-enabled it. To be successful in such an endeavor, the user would have to remember, when writing the second file, exactly how many breakpoints would be set once the first file was **source**d into the debugger. Thus gdb's restriction on the syntactic form of the **enable** and **disable** commands' parameter destroyed the user's ability to write *independent* debugging code in different files, and to **source** those files into the debugger in any order.

Dalek, through a minor syntactic extension that was easily implemented, allows breakpoints to have mnemonic "names". Dalek maintains a read-only global convenience variable, "$LAST_BP_SET", designating the last breakpoint set. Typically it contains the *number* of the last breakpoint that was set. Thus, by practicing the coding convention exemplified by Figure 2-5, the user can easily write *independent* debugging routines, located in different **source** files, that utilize the **enable** and **disable** commands.

To be fair, in gdb it would be possible to use similarly mnemonic global variables as Boolean flags. For example, the code in Figure 2-6 would achieve the same result under gdb as that of Figure 2-5 would under Dalek. Yet if *foo*() were invoked a thousand times by the inferior before *goo*() were ever called, the difference in execution times would be enormous. The only breakpoint "naming" convention permitted under gdb would entail two thousand extra UNIX context switches and one thousand extra tests by the debugger of a convenience variable before the commands attached to *foo*()'s breakpoint could be performed. This is because, as mentioned in Chapter 0, the term "conditional

```
# This example shows how to ensure that a breakpoint in foo will
# remain disabled until the inferior has executed procedure goo once.

break foo
set  $foos_bp = $LAST_BP_SET
#   If a breakpoint is to be "named" as above,
#   this should be done before any other breakpoints are set.
disable  $foos_bp
commands                # Default parameter is $LAST_BP_SET
   . . .                # Specify the command block.
end

break goo
set  $goos_bp = $LAST_BP_SET
#   Dalek automatically updates $LAST_BP_SET to reflect the number
#   of the last breakpoint set by the user.
commands                # Default parameter is $LAST_BP_SET
   enable  $foos_bp     # Enable the breakpoint in foo by name.
   disable  $goos_bp    # This breakpoint will disable itself when it executes.
   cont
end
```

Fig. 2-5 Example of Dalek coding convention utilizing breakpoint ‘names’.

Assign a "name" to the breakpoint in *foo*(). Give it a Boolean
value indicating that the breakpoint should *pretend* to be disabled.
set $foos_bp = 0
Now set a *conditional* breakpoint in *foo*()
break *foo* if $foos_bp
commands
 . . . # Specify the command block.
end

break *goo*
commands
 set $foos_bp = 1 # **"**Enable**"** the breakpoint in *foo*.
 cont
end

Fig. 2-6 An *inefficient* way gdb could use breakpoint ‘names’.

breakpoint" is a misnomer. When the inferior encounters such a breakpoint, it will always execute it, but then the debugger will quietly transfer control back to the inferior without notifying the user.

The relaxation on the syntactic *form* of the parameter to **enable** or **disable** provides more than just a superficial benefit to the user. In and of itself, this extension already frees the user to discover and adopt more powerful debugging methods by using breakpoint names.

But this is not yet the end of this particular story. We stated earlier that

> Dalek maintains a read-only global convenience variable, "$LAST_BP_SET", designating the last breakpoint set.

In light of the user’s ability to broadcast an entire *set* of breakpoints with one Dalek command, it might seem that our choice of words in the above passage was a trifle ambiguous. In fact, the $LAST_BP_SET feature has also been thoroughly integrated with other elements in Dalek's toolset. For example, immediately after the user issues a **proc-break** command, $LAST_BP_SET will contain not a single *number*, but rather a *string* of numbers — a list of every breakpoint number in the newly broadcast set. Thus, as shown in Figure 2-7, the user can easily manipulate an entire *set* of breakpoints (broadcast or otherwise) with a single command.[5] Note that this mechanism places no structural constraints on the user, allowing him to name and manipulate a "set" of breakpoints whose membership is completely independent of that of any underlying broadcast

```
proc-break  <regular-expression-1>   # Broadcast a set of breakpoints.
    . . .        # Specify the shared command block for this set.
end

# Next, make a permanent record of the "names" of all members of this set:
set  $regexp1_set = $LAST_BP_SET
disable  $regexp1_set        # Temporarily disable all members of this set.

set  $regexp1_subset1  =  "4, 5, 6, 79, 80"     # Specify a subset.
set  $regexp1_subset1_union_regexp2 = "$regexp1_subset1, $regexp2_set"
                                               # Form a union!
enable  $regexp1_subset1_union_regexp2       # Selectively enable that union.
```

Fig. 2-7 Perhaps we should have named it '$LAST_BP_SET_set' ?

[5] Another natural extension would be to allow the **commands** command to accept such a set as its parameter, or to make an entire set of breakpoints *conditional* with one command. Although these features are certainly possible, they have not yet been fully implemented.

(or singleton) sets. The defining characteristic of a true broadcast set is that all its surviving members (i.e., those original members who have neither been deleted nor been assigned a new command block) share a *physically common* command block.

As so often happens in software toolbuilding, once a syntactic form has adapted to more naturally express a desired functionality, this can cause the underlying semantics to undergo a corresponding evolution. In this case, we first progressed to two different styles of names for the same underlying type, and thence to two entirely different underlying types.