CHAPTER 5

Design and Implementation Highlights

After some legal preliminaries, this chapter first offers a brief tour through the implementation of gdb, the GNU debugger upon which Dalek was based. We then describe in fair (but not excruciating) detail how each of Dalek's major new features was realized in the implementation. Although this chapter will undoubtedly be of interest to implementors of related systems, significant portions will also be of interest to other, more general readers. We emphasize that the later sections do *not* constitute a low-level, line-by-line walkthrough of Dalek's source code. Rather, they review some of the design decisions and tradeoffs that were made, and the resulting effects on the structure of Dalek's source code. Various significant semantic issues that were elicited during the design or implementation phases are examined in Sections 5.4.1.1, 5.4.3.1, and 5.4.7.

5.1. Naming and Disclaiming

"Dalek", in the legal sense, consists of C language source code that is useful and meaningful only when viewed as a collection of additions, deletions, and changes to gdb, another body of C code copyrighted by Richard Stallman and the Free Software Foundation (FSF) of Cambridge, Mass. FSF publicly distributes source code with the proviso that any party is allowed to modify the source and subsequently distribute it, so long as all such transactions are essentially profitless.

It may be helpful at the outset to clarify our usage of the names "Dalek" and "gdb", as they are typically used in a *non*-legal sense in this and other chapters. For purposes of convenience, "Dalek" refers to a complete program and to the underlying C source code that, when compiled, forms that program. This source code includes portions written and copyrighted by the FSF, portions copyrighted but not written by the FSF, as well as portions written by the author and W. Wilson Ho that were subsequently contributed/released to the public domain by the Regents of the University of California.

It should be understood that by thus incorrectly enlarging the referent of the name "Dalek", our intention is solely to make the language of discussion and comparison less awkward. It does *not* constitute a claim that the legal referent of "Dalek" is identical with this informal referent used throughout this paper. Having dispensed with the disclaimers, on to the discussion.

5.2. Version History

Any attempt to describe the implementation of gdb is complicated by the fact that, as with many large programs, gdb continually undergoes revisions — some of which entail nontrivial structural modifications. Dalek was originally based on version 2.6 of gdb, which was first released by FSF for public distribution on May 5, 1988. Version 2.7 followed in September 1988, and because at that stage the Dalek code was so rudimentary that all except a few lines resided in separate files from the gdb source code, it was possible to automatically upgrade Dalek using the UNIX *patch* utility.

In December 1988 when version 3.0 of gdb was released, it was not worthwhile to immediately attempt to upgrade Dalek. Our reluctance stemmed from the fact that by this time, a large part of Dalek's code had become tightly interwoven with gdb's source code in order to fine-tune the behavior of all the interacting features. Because our attempts to use *patch* for this upgrade resulted in so many rejected chunks, we suspected that even laboriously doing it by hand would probably introduce new bugs.

Eventually in June 1989, after Dalek was relatively stable, we undertook to upgrade it to gdb version 3.0. Two considerations motivated this decision. First, we wanted to incorporate into Dalek a bundle of fixes to pre-existing bugs that had been inherited from gdb-2.7. Second, we felt that to be of widespread utility, Dalek had to keep current. Since we hoped to one day transfer the honor of maintaining and upgrading Dalek to the FSF, it was in our interest to make that transfer as painless as possible. Giving the FSF a collection of changes to version 2.7 would not be terribly valuable if that occurred when the current version of gdb had advanced to 9.2.

In February 1990 we made the latest code for Dalek available to the FSF. They are considering re-integrating Dalek with the latest version of gdb (a Beta test version of which has now evolved to 3.5) and distributing the result as gdb-4.0.

Gdb runs on a surprisingly wide range of machine architectures including machines based on the Intel 386, Motorola 680x0, VAX architecture, etc. Gdb's source code reveals that periodic attempts have been made to maintain a dichotomy between the machine-dependent and machine-independent parts. Where possible, machine-dependent routines are placed in separate files. Since strictly adhering to this practice would lead to an ungainly proliferation of files, in-line conditional compilation has also been used in some sections of the otherwise "machine-independent" files. This chapter will attempt to concentrate on the machine *independent* differences between Dalek and gdb.

5.3. A Brief Tour of Gdb's Implementation

5.3.1. Gdb's Basic Interpretive Structure

At the highest level, the structure of gdb is simply that of a line-oriented interpreter running a read-eval-print loop to interact with the user. Many of the commands in the debugger language, e.g., **print** or **backtrace**, fit quite readily into this read-eval-print paradigm. Yet 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.

Recall that under UNIX, the inferior runs as a separate process. Thus a command like **cont** will ultimately make recourse to the UNIX system call *ptrace*() to inform UNIX that the inferior process may be placed back in the operating system's ready queue. Although the other commands like **step** or **call** can be conceptually lumped with **cont**, the details of their actual implementations in some cases depend not only on specific machine architectures, but also on the version of gdb in question. Wherever possible in this discussion, we will try to gloss over such details.

Whenever a command like **cont** allows a resumption of the inferior process, gdb must then delay — via the UNIX *wait*() system call — until the inferior's execution terminates or is suspended. If the inferior terminates normally, gdb will be so notified by UNIX. Alternatively, if the inferior encounters an abnormal condition such as hitting a breakpoint or receiving a signal, UNIX will suspend the inferior and return control to gdb. Of course, gdb must be able to distinguish among these various cases. For example, if the inferior hit a breakpoint that had a block of debugger commands associated with it, then gdb must know to interpret those commands before returning to the outer, interactive loop.

5.3.2. Debugger Command Blocks in Gdb

Gdb has a second generic ability whose implementation has influenced its mid-level structure. In certain circumstances, gdb will read in a block or sequence of debugger commands, and store that block in memory. Later, if and when appropriate, the commands in the block will be interpreted sequentially.

In gdb, a command block can only occur in one of two ways: it may be associated with a breakpoint, or it may be bound to a user-defined command (essentially an abbreviation for that sequence of debugger commands). A block of commands is implemented in gdb as a singly-linked list of structs called "command_lines".

Although an apparently similar situation occurs when the user orders gdb to **source** in a sequence of debugger commands from an external file, that action is actually accomplished by recursively calling another read-eval-print loop whose input is redirected from that file versus from the standard input.

5.4. Major Differences between the Gdb and Dalek Implementations

The following might loosely be considered a design review. Yet the reader should bear in mind that Dalek's development has deviated considerably from an idealized, requirements-driven process. We began with gdb, a program we initially did not understand very well. We occasionally implemented features whose specifications were planned in advance, but just as often we were driven only by vague feelings that things should *somehow* be improved. Many of our ideas formed and evolved only as we gained experience actually using Dalek to explore and debug programs. A significant portion of the development effort was spent in getting various features to work well together, so they would properly DWIM (Do What I Mean) in a number of diverse debugging contexts.

Obviously, much of our development effort was specific to the nature of the code for gdb, Dalek's underlying platform. If Dalek had instead been based on a different (albeit conventional) debugger, the specifics of its implementation would undoubtedly be different. Nevertheless, the flavor of the following sections would probably have remained fairly constant. On the other hand, if Dalek's underlying platform had been radically different — e.g., had that platform provided a rich, virtual-machine environment to support interpreted rather than compiled C code — then not only Dalek's implementation, but perhaps even its feature set might differ significantly.

5.4.1. The 'if' and 'while' Statements

Historically the **if** and **while** commands were among the first significant new features implemented in Dalek. Pursuant to our philosophy of making our enhancements as compatible as possible with gdb's existing code and data structures, the textual body of an **if** or **while** statement was represented internally as a linked list of command_lines.

Because of the need to influence control flow (e.g., looping by the **while** command, and conditional interpretation including the handling of optional **else** clauses by the **if** command), two new routines were written to achieve the desired functionality. Fortunately these new routines, do_if() and do_while(), were relatively autonomous, that is, they did not require major alterations to other parts of gdb's source code.

Gdb, like most interpreters, is inherently line-based. To avoid extra development work and to maintain compatibility with future versions of gdb, we elected not to change this strategy. The fact that the **if** and **while** statements were the first commands requiring conditional branches across multiple lines was accommodated rather easily. However, this line-oriented design caused a slight difficulty because do_if() and do_while(), depending on the context in which they are called, may need to look in different places for the commands comprising the bodies of their respective statements. For example, if the user types the first line of a **while** statement interactively, then once the carriage return is typed, Dalek's primary interpretive loop will go into action scanning the text of that line. Recognition of "**while**" as a keyword causes do_while() to be called, and it must then read subsequent commands from the standard input, up until the matching **endwhile** is typed. Clearly it would be counterproductive if an invocation of do_while() for an inner, nested **while** statement attempted to read its body in a similar manner. To do so would mean that once the user had finished typing the bodies of both **while** statements, and interpretation had proceeded to the inner, nested **while**, Dalek would at that point then wait for the user to begin typing the body for that inner **while** statement.

The program logic to distinguish these situations also needed to recognize when the entire **if** or **while** statement was itself already embedded in another block of commands — e.g., in a block of breakpoint commands or in the text of a user-defined function. Further complicating the design was the possibility that the user might **source** in commands from an external file, and that the file might itself contain an **if** statement to conditionally **source** in commands from yet another file.

The solution was surprisingly simple — two new global variables were used to indicate whether Dalek was currently interpreting commands syntactically within the body of an **if** or **while** statement, although one Boolean global would have sufficed for this purpose. To locate the bodies of their respective statements, do_if() and do_while() first check this status information. If their allotted statement is statically nested within another **if** or **while** statement, then their statement's body was already read into memory. Otherwise, if they find themselves to be the outermost such statement, they next determine whether they are responsible for reading the body of the statement into memory — either from standard input or perhaps from a **source** file. Should neither of the previous conditions hold, then the **if** or **while** statement must itself already be embedded in a list of command_lines. In this case, as in the first alternative, the body of the statement can be found in a singly-linked list immediately following the command_line containing the text of the **if** or **while** statement's conditional expression.

5.4.1.1. A Semantic Issue Regarding the 'while' Statement

An interesting semantic issue cropped up after we had gained some experience as users of Dalek's **while** statement. This is the same concern that we hinted at during our discussion of Figure 3-2. To understand this issue, it will first be necessary to give more detail regarding the internal structure of command_lines. A struct command_line contains three fields: a meta_break flag, a pointer to the next command_line in the singly-linked list, and a pointer to a character string that contains the actual text of the debugging command. Thus command blocks in gdb are strictly interpreted, i.e., no attempt is made to "compile" them into a more efficient internal representation.

Gdb does, however, have an efficient internal representation for expressions, and also provides a routine to evaluate them. Initially, we decided to "compile" the text of a **while** loop's conditional expression into this internal form, anticipating that since the loop test was likely to be evaluated several times, this design might allow a significant speedup during execution. But in certain situations, this practice ran afoul of the Principle of Least Surprise.

One case in which "compiling" a conditional expression leads to unexpected behavior is where the inferior process is suspended within a procedure having a local variable, say *foobar*, and the user wants to continue executing in that procedure until *foobar* becomes negative. The user might compose the simple watch loop of Figure 5-1 to achieve this result.

If Dalek compiles this loop test into gdb's more efficient internal form, no reference to the name *foobar* will remain. Instead, the internal form will instruct Dalek to compare an integer located in the inferior at, e.g., address 0x203a with zero. Now if *foobar* remains

```
while ( foobar >= 0 )
   step
endwhile
```

Fig. 5-1 Simplistic code for watching a local inferior variable.

non-negative throughout the execution of its defining procedure, Dalek will continue to single-step the inferior even after that procedure returns to its caller. Not until the inferior's runtime stack once again grows so that an arbitrary procedure's stack frame encompasses the former site of *foobar* could this "optimized" implementation of do_while() allow a chance to exit the loop, and inform the user that the value of *foobar* has changed. Needless to say, such behavior would profoundly mislead the user.

Because we felt it would be unrealistic to expect even naive users always to code such loops in the preferred style of Figure 5-2, we decided to forgo the dubious benefits of compiling the loop expression. Note, however, that even with Dalek *not* "optimizing" the loop test, the simplistic coding style of Figure 5-1 could still cause misunderstandings. For example, should the inferior program also contain a *global* variable named *foobar*, then that would become visible to Dalek as soon as execution departed from the scope defining the local *foobar*, unless, equally hazardous, a new function were called that *also* declared a local *foobar*. In such cases, the simplistic debugger code of Figure 5-1 might continue to single-step the inferior program long after exiting the current procedure, until the value of the global *foobar* became negative. Alternatively, it might be fooled into terminating prematurely if the new local masking the original referent of "*foobar*" ever became negative. The coding conventions illustrated in Figure 5-2 will normally suffice to prevent misunderstandings regarding the referents of global names as well as local names, since use of the **next** command *typically* ensures that the condition the user wants to monitor will only be checked at intermittent points during the inferior's execution, namely, *exactly* those points where its locus of execution intersects the watching loop's initial "isobar" or plane of constant scope in the inferior's contour model. But

{
local $current_proc
set $current_proc = **$func**(0)
while (*foobar* >= 0 && $current_proc == **$func**(0))
  next # *Not* **step**, since control might stray beyond the local scope.
endwhile
}

Fig. 5-2 A safer coding style for watching a local inferior variable.

the devil is in the details, and nowhere is this more true than in debugging. For the record, this particular devil is one (relatively benevolent) instance of what we call a "loosely-bound" watch loop. These will be discussed ad nauseam in Section 6.4.2.1.1.

Note that, whether a global or local entity is being watched, the user may not be satisfied with such intermittent monitoring, but may desire a finer granularity — taking strides that are either the length of source code lines or of individual machine instructions. In such a case, he should avoid the use of potentially ambiguous names within the loop altogether. Rather, a new local debugging variable(s) should be declared and, before entering the loop, the user should, e.g., **set** $foobar_pointer = (int *) &*foobar*. The loop condition, and all statements within that loop, should thereafter refer, e.g., to "*$foobar_pointer" rather than to "*foobar*". When taking **next**-sized strides, such a method would also guarantee the proper semantics under *all* circumstances — even those in which execution crossed a scope contour by entering a nested block within the same, original inferior procedure, in which new local variables with conflicting names were declared. Of course, for a register variable, whose "home address" may change any time execution crosses a scope contour, this technique is not viable. We also choose to ignore any "helpful" optimizations the compiler may have performed on our behalf.

In situations where a loop is maintaining constant surveillance over a *local* entity by taking tiny **stepi**'s, Figure 5-2 would require one more surgical alteration in addition to replacing all occurrences of, e.g., "*foobar*" by "*$foobar_pointer". Notice that its loop termination test would no longer be adequate because the loop would prematurely terminate the first time $current_proc invoked another procedure. Thus the loop test should not compare $current_proc to **$func**(0), but might instead employ **$t_frame**() to compare the relative height of the inferior's current runtime stack to that of its initial value, stored into yet another local debugging variable just before entering the loop. For example,

```
local $init_stack_height
set $init_stack_height = $t_frame()
while  ( *$foobar_pointer >= 0   &&  $init_stack_height <= $t_frame() )
    stepi
endwhile
```

5.4.2. Events, Local Variables, and User-Defined Functions

As with the **if** and **while** commands, we were able to implement events, support for local debugger variables, and user-defined functions, without extensive modifications to gdb's pre-existing source code. Although the code implementing these new features was essentially self-contained and isolated in separate files, it did require several "hooks" into the lexical scanner so that names were resolved differently during expression parsing than they had been formerly.

5.4.3. Stacking of ‘Pending’ Command Blocks

Recall that in gdb, a block of commands was represented as a singly-linked list of struct command_lines, and could exist only under two circumstances: either the commands were bound to a breakpoint or they constituted a user-defined command. In either context, the routine responsible for "driving" interpretation through the linked list could use a local variable to point to each respective command_line in the sequence, advancing that local pointer as each command was interpreted. But with the advent of multi-line statements like **if** and **while**, the old, line-oriented algorithm was no longer adequate because the responsible "driver" would need to advance its local pointer not to the next command_line, but rather to the command_line following the matching **endif** or **endwhile**. One solution was for the "driving" routines for breakpoint commands or user-defined commands to continue using a local variable to traverse their respective lists of command_lines. Should they encounter an **if** or **while** command, they would then need to scan the text of each subsequent command_line in the linked list in order to find the appropriate matching **endif** or **endwhile**. Although inefficient, this method would have been able to handle multi-line statements such as **if** and **while**. But it would have been extremely messy to integrate this method with support for Dalek’s "introspection" facilities.[1]

Instead, we chose to make the formerly local command_line pointers global: A single global pointer is now used to indicate the next embedded command_line to interpret in a

[1] Specifically, Dalek’s **dtrace** command needs to display the next pending command_line in all stacked command blocks. In addition, the user might **edit** a pending command_line. If the new text were longer than the previous contents, memory for the new string would need to be allocated. Then the *local* variable (belonging to a "driving" routine somewhere below in Dalek’s own runtime stack) that pointed to the old command_line would need to be redirected.

given block.

As so often happens, solving one problem led to (or, more accurately in this case, exacerbated) another — we now had to handle cases where part way through one block of commands, interpretation of another block was initiated. One example of such a case would be, within a block of breakpoint commands, to embed a command raising an event. Another common way for such a situation to arise would be, within the text of a user-defined function, to embed a **step** or **jump** command, since resuming execution of the inferior process might cause it to hit a breakpoint to which a block of breakpoint commands is bound. The natural solution was to allocate a global stack on which a pointer to the next pending command_line in the original block could be pushed as the focus of interpretation shifted to the command_lines comprising the (dynamically) inner block.

A simple implementation of this method was adequate for Dalek's initial, very primitive attempts at user-friendliness. But more elaborate code was ultimately required to push and pop not only the value of the global command_line pointer, but also information regarding Dalek's current context: For example, Dalek has evolved so that it now must maintain meta-debugging information regarding every stacked block's type and name. It must also track for each block, whether the pending locus of interpretation is syntactically inside the body of an **if** or **while** statement, so that the two global variables mentioned in Section 5.4.1 can be refreshed whenever interpretation within a block concludes and control returns to its dynamic caller.

It should be emphasized that conflicts regarding the treatment of dynamically nested blocks of debugger commands already appeared in gdb in more than merely embryonic form. Gdb's convention of discarding all breakpoint commands syntactically following a **cont** or similar command was typically only a minor inconvenience to the user. In contrast, when interpretation of any command_line embedded in a block caused the *invocation* of an inferior procedure containing a breakpoint, not only would all subsequent commands in the original block be disregarded, but in addition no commands associated with the breakpoint in the called procedure would be interpreted either. Having exceeded its level of competence, gdb would give up with the despairing message, "Cannot continue previously requested operation. Unable to restore previously selected frame". This behavior was decidedly unfriendly and unnecessarily restricted the user's First Amendment rights, violating not only the doctrine of "Do What I Mean", but even that of "Do What I Say, Damnit".

5.4.3.1. Semantic Issues Prompted by Stacking of 'Pending' Commands

Software implementation, like hanging, has a wonderful ability to clarify and concentrate one's thoughts. It was not until this stage in Dalek's evolution that we realized several important semantic issues were lurking nearby. For instance, suppose Dalek is interpreting commands that are statically inside a **while** loop when it encounters a call to a user-defined function. Dalek will push information regarding the **while** loop's command block on its global stack and then begin interpreting commands in the body of the user-defined function. Now suppose the body of the user-defined function contains a **wbreak** command that is not syntactically inside a **while** loop of its own. How should Dalek respond? Should it immediately cease interpreting the user-defined function and proceed with the next statement following the *dynamically* enclosing **while** loop — as though an exception had occurred? What about the value to be returned by that user-defined function? Alternatively, should interpretation of the user-defined function proceed normally, so that the **wbreak** command is only heeded once Dalek is poised to resume interpretation of the **while** loop? Should this latter alternative imply that a **while** loop can itself return a value? A plethora of related issues springs immediately to mind, e.g., can a user-defined function cause processing of a dynamically enclosing event to abort by giving the **event exit** command?

We decided in most cases to prohibit such *specific* interference in the internal affairs of another block, but with notable exceptions. One such exception was to permit the user to **close** the current **source** file even though that **close** command was issued from an arbitrarily deep nesting of dynamic blocks of various types. Another exception to this rule was made to allow user-defined commands to engage in this "up-level" meddling. The justification was that since a user-defined command is supposedly an abbreviation for a sequence of commands, it is appropriate to treat it as a macro whose code is expanded inline. Viewed in this fashion, e.g., a **wbreak** command inside a user-defined command *would* appear to be statically within an enclosing **while** loop.[2]

Another exception to the rule prohibiting *specific* up-level meddling was made to allow the user, while in **pause** mode, to issue commands altering control in the nearest dynamically enclosing block. This was justified on grounds of user-friendliness. Such commands are not carried out immediately, but instead are deferred until the user issues an **unpause** command. In this way, the user can **wbreak** out of a **while** loop, **func-return** a

---

[2] That Dalek's implementation does *not* actually expand user-defined commands into inline code is all too apparent. For example, if one specifies a user-defined command containing an **else** clause without a surrounding **if . . . endif**, and then attempts to use that user-defined command within an **if** statement, an error will result.

value from a user-defined function, or **event exit** to terminate the processing of the enclosing event's command block.

Having unleashed prospective users in this potential wilderness of dynamically nested blocks, we thought it only fair to supply a *generic* command for bailing out of an arbitrary number and type of dynamically enclosing command blocks. The **set exit N** command will do just that — it will immediately pop **N** command blocks off Dalek's global stack, and interpretation will resume with the next pending command in what was formerly block **N**. As will be seen in Figure 5-4 in the next section, this generic ability to exit from any type of command block provides a solution to a completely unrelated problem (tail recursion).

For consistency with the labelling conventions used by Dalek's **dtrace** display, some "virtual command blocks" are counted and "popped" by the **set exit** command as well. The clearest case is where the user can exit from a multiply **pause**d state, even though "[PAUSE]" is obviously not the same sort of animal as a command block. Other "virtual command blocks" include those in Figure 4-6 labelled

> [FINISH *has_cycle*] and
> [STACK_DUMMY *has_cycle*]

The latter indicates that the user has **call**ed an inferior function directly from Dalek — necessitating the construction of a dummy frame on the inferior's runtime stack — and the former indicates that the inferior has been ordered to **finish** or **$quiet_finish**() its current procedure. Although a command block is not associated with either of these labels, they serve as important placeholders, reminding the user of the forces he has set in motion. Experience has demonstrated that without their presence in a **dtrace** display, it is difficult for the user to comprehend Dalek's status. That, in turn, could cause the user to inadvertently corrupt the inferior's runtime stack in some cases.

5.4.4. Elimination of Tail Recursion

Earlier, we claimed to have solved the problem of suspending interpretation within one block of debugger commands in order to begin interpreting a dynamically nested command block by maintaining a global stack of pointers to pending command_lines. This allowed the user, for example, to place a **step** command midway through a block of breakpoint commands, and yet retain confidence that subsequent debugging commands in that block would indeed be interpreted after the **step** had completed.

Since we had followed the most straightforward route in implementing this behavior for Dalek, one regrettable implication was that our design was susceptible to tail recursion. Probably the most diabolical example of tail recursion imaginable is unfortunately also one of the most common situations encountered in debugging: the user sets several breakpoints in the inferior program, each of which has a block of breakpoint commands ending in **cont**.

Using Dalek in such a situation would result in a buildup on Dalek's own UNIX runtime stack of activation records for calls to do_breakpoint_commands(). Each such call would have just interpreted a **cont** debugger command as the last command_line in its assigned sequence. When the flow of control eventually returned to any such calling frame, that invocation of do_breakpoint_commands() would discover that it had finished interpreting its sequence of command_lines, and would promptly return. However, until the inferior process terminated or hit a breakpoint whose command block did not contain a **cont** command (thus allowing the call chain to unwind), Dalek's own UNIX runtime stack would grow monotonically.

This behavior would be acceptable in some circumstances — those in which the user had inserted only a few breakpoints in the inferior program, and none were inside loops. Yet we anticipated (and have encountered in practice) debugging situations, especially those arising from the broadcast modes, in which the user would employ a vast number of breakpoints. In those cases, we would run a significant risk of Dalek overflowing its own UNIX runtime stack.

Although in principle tail recursion is easy to eliminate, several factors complicated this undertaking in Dalek's case. Chief among them was the fact that Dalek uses several different procedures to drive its interpretation through different types of command blocks — for example, it has one routine dedicated to user-defined functions and another specifically for events. These (indirectly) mutually recursive routines would either need to be combined into one very general routine (a major headache) or else would need to carefully follow an exact protocol in their dealings with each other.

But even if this were done, two formidable problems would remain before tail recursion could be eliminated in all cases. One problem is that do_if(), the routine driving interpretation through the body of an **if** statement, would need to recognize when it encountered the "last" command_line in the body of its statement, in order to defer interpretation of that command to whatever driver was interpreting the block of debugger commands that dynamically enclosed the block containing that **if** statement.

Suppose the "last" command_line to be interpreted in a particular **if** statement immediately precedes the **else** clause. Then not only must the enclosing driver know to interpret that command_line, but it also must know that afterwards it should skip forward in the linked list of command_lines until it finds the matching **endif**.

And consider the syntactically last command_line in the body of a **while** loop. Since interpreting that command might change the result of evaluating the **while** loop's conditional expression, how are we to know in advance whether that command_line will *dynamically* be the last one interpreted in the **while** loop? And of course, someone must see to it that that final evaluation of the **while** loop's conditional expression actually *is* carried out, in case it has important side effects. Even then, the issue of how to deal with the **wbreak** command would still remain to be resolved.

The other recalcitrant factor to be addressed to achieve a general solution to tail recursion in Dalek's implementation is interaction with the error handling and the meta-debugging features. For reasons that will become clear in later sections, friendly error handling and accurate meta-debugging support require that certain housekeeping chores be performed as each "driving" routine is exited. For the moment, suffice it to say that a general solution would be quite intricate.

Since the most pathological case of tail recursion Dalek was likely to encounter — that of interpreting a series of breakpoint command blocks, each ending in a **cont** command — was also the only one expected to occur frequently in practice, we chose to eliminate tail recursion only for that particular situation. Even so, integrating this solution with the error handling and the meta-debugging features was fairly involved. For example, when do_breakpoint_commands() "prematurely" exits, ceding interpretation of the last breakpoint command to the routine that is driving Dalek’s interpretation through the dynamically enclosing command block, it must *not* perform the normal housekeeping chore of popping (from Dalek’s internal, global stack) the type and name (e.g., "[BP#3]") of the command block it has been interpreting. To do so would mislead the user (in **dstep** mode, or via the **dtrace** command) into thinking that this last breakpoint command was actually embedded not in a block of breakpoint commands, but rather in the dynamically enclosing command block itself.

One implication of this method of eliminating tail-recursion is that it penalizes what might seem to be the most straightforward way to code a *conditional* **cont** in a block of breakpoint commands. For example, the code fragment in Figure 5-3 incorporates tail-recursiveness in a manner that Dalek’s implementation unfortunately does not recognize. The preferred coding convention is shown in Figure 5-4.

```
commands
    . . .   # Main body of command block.
  if  ( $whatever_condition )
    cont     # Continue the inferior process.
  endif
end    # End of block of breakpoint commands
```

Fig. 5-3 Possibly *unsound* coding style for continuing the inferior.

```
commands
   . . .        # Main body of command block.
  if  ( ! $whatever_condition )
    set exit 1    # Avoids the following cont command.
  endif
  cont     # Continue the inferior process.
end    # End of block of breakpoint commands
```

Fig. 5-4 *Preferred* coding convention for continuing the inferior.

5.4.5. Error Handling

Since Dalek allows (nay, encourages) the user to write and use block-structured debugging commands, some thought must be given to the error recovery strategy, especially since an error may occur deeply nested in a dynamic chain of command blocks. Simple examples of such errors might include misspelling a command or the name of an event. Another common error would be to attempt to reference an inferior variable by an unknown name relative to the inferior process's then-current scope.

Gdb's solution upon encountering an error was simply to return to interactive mode, discarding any "pending" commands. This behavior was generally acceptable, since it was impossible for the user to become deeply entangled in any nested blocks of debugging commands. However, for Dalek to continue that practice would not only have imposed an occasional heavy burden on the user, but more importantly, it would have had a chilling effect on the user's inclination to write block-structured debugging commands in the first place, knowing of the enormous penalty for even one small error.

Our initial error handling strategy for Dalek was to practice incremental error recovery: That is, when encountering an error in a dynamically nested block of debugging commands, Dalek would first notify the user and display the offending command's text, and would then present the user with several choices of what action to take next. The mildest option was just to proceed to the next command in the current block. Barring that, the user could elect to discard all remaining commands in the current block. For the most severe errors, all pending commands on the entire dynamic call chain could be discarded. This capability was achieved partly by copious use of the UNIX setjmp() and longjmp() library routines.

However, even brief experience as Dalek users convinced us that this initial strategy was inadequate — in particular, because the most friendly and useful option had not been offered to the user at all: that of correcting the offending command before proceeding to interpret any subsequent commands in the block.

During this stage in Dalek's development, our primary concern was to prototype a research vehicle. User-friendliness was considered a secondary goal, and we were disinclined to undertake any massive development effort merely to implement some feature that was already well-understood and commonplace in other debuggers or interpretive systems.

Thus although we recognized the desirability of having some facility for editing embedded commands "on the fly", we chose not to implement one at that time. Instead, we pursued a more versatile feature: the ability to **pause** at any point during the interpretation of a block of debugger commands, interpret an arbitrary number of interactively-typed commands, then to **unpause** and resume interpreting at the next pending command_line.

This ability allowed Dalek to be reasonably friendly with respect to handling the more common types of errors. Once the text of the offending command was displayed, the user could first enter whatever commands were required in order to fully understand the error. If the name of an event were misspelled, the user could ask Dalek to display the names of all currently defined events. The user could then type in the correct command and **unpause** Dalek, whereupon it would proceed to interpret the remaining embedded command_lines.

With the option to **pause** being offered as a choice by the error handling routine, Dalek was almost as friendly in this regard as it would have been had it instead provided a facility for editing embedded commands. (Of course, errors that occurred within loops were still annoying.) However, had we implemented a facility for merely editing embedded commands, that would not have conferred upon the user the primary benefit of the **pause** option — the ability to "poke around" and to explore both the context in which the inferior process is currently suspended, as well as the dynamic environment in which the debugger finds itself.

As an example of the usefulness of knowing the debugger's own context, suppose the text of the offending embedded command were "**set** $x += 2", and the error message said, "Argument to arithmetic operation not a number".

Now it is possible that that same command appears in several different command blocks the user has constructed. Which one of them is the current block? Is the problem that "$x" should instead read "$y"; or is it perhaps that "$x" was never initialized with a numeric value and hence is of type VOID?

Section 4.2, on meta-debugging, treated the "introspection" features of Dalek that allow it to communicate its own state to the user in meaningful terms.

As an added illustration of the versatility of **pause** mode, we were easily able to implement a command by the same name, thus allowing users to include a **pause** command in every block with which they anticipated potential problems, or whose dynamic environment they desired to explore.[3] In retrospect, of more importance than our decision to implement a command to **pause** was a different mission for **pause** mode that came essentially "for free" with the source code inherited from gdb. Gdb already contained a short interrupt handler that could call its error routine whenever the user typed Control-C. With only minor modifications, the user could now enter **pause** mode by typing Control-C interactively. This proved especially useful to abort or silence certain commands that generated a substantial amount of output, or to interrupt Dalek in the midst of an interminable **while** loop. At that point, the user could then inspect the value of the loop index as an indication of when or if the termination condition might ever be met, and might opt to manually break out of that loop so that once **unpause**d, Dalek would proceed to interpret the next embedded command following that loop.

5.4.6. Broadcast Breakpoints

Recall from Sections 0.3.2, 2.2.1 (i.e., Figure 2-1), and 2.2.3 (i.e., Figure 2-7) that Dalek allows the user to disseminate or "broadcast" breakpoints to many different locations in

[3] This cumbersome usage of proliferating **pause** commands was later superseded by meta-breakpoints, although the implementation of meta-breakpoints ultimately still calls the original pause() routine.

the inferior process by issuing just a single command. All breakpoints in such a broadcast set initially share a common block of breakpoint commands specified by the user at the time that that set of breakpoints is broadcast.

Although broadcasting breakpoints in either mode will cause the user's newly specified block of commands to be associated with every breakpoint in the new set initially, the user is free to subsequently assign different blocks of breakpoint commands to individual members of such a set. This will not affect the command block for the breakpoints remaining in that broadcast set.

Implementation of broadcast breakpoints (for both modes, whether the "target" is an opcode or a regular expression) was surprisingly easy, since gdb already provided many of the lower-level procedures required to support this functionality. The most involved part was the maintenance of "reference count" data so that, if and when every breakpoint in a broadcast set was either deleted or assigned a different command block, it would be clear that the linked-list of command_lines for that broadcast set could be freed. This necessitated the addition of a ref_count_index field to gdb's previous definition of a struct breakpoint. Thus every breakpoint now has an associated index that, if nonzero, is considered to "point" into the array pb_ref_count[].

At every place in Dalek's source code where a breakpoint will be deleted or have its command block redefined, if its ref_count_index field is nonzero, then the contents of, e.g., pb_ref_count[breakpoint−>ref_count_index−1], will be decremented. The contents of that slot in the array indicate how many breakpoints share the same command block. Thus if the new contents are zero, this indicates that it is safe to free the command_lines in the linked list attached to the newly-deleted or redefined breakpoint.

5.4.6.1. Efficiency Considerations for Broadcast Breakpoints

Now in standard gdb (all versions through gdb-3.4), whenever control was transferred from an inferior back to the debugger, gdb would temporarily remove *all* breakpoint instructions from the inferior, replacing them with their original respective machine code instructions.

Presumably this was done so that if the user asked for an assembly language listing of a section of code, the breakpoint instructions would not appear. Had they been printed out, the result would typically be very misleading because wherever a breakpoint instruction (invariably one of the shortest in any instruction set) had overwritten a longer instruction, the disassembled listing would exhibit a "framing error"; i.e., the latter section of a partially-overwritten instruction would be interpreted as the beginning of a new instruction.

We felt that to continue this policy under typical usage conditions in Dalek would be to incur a potentially enormous speed penalty, since each *ptrace*() call required to remove a breakpoint instruction would force the UNIX kernel to perform two context switches. Although context switches are not logically necessary under these circumstances, most implementations of UNIX maintain the inviolability of a process's address space by mandating that all requests by a debugger to read or write an inferior process's address space be channeled through the *ptrace*() system call. *ptrace*() effectively forces the UNIX kernel to first wake up the inferior process and induce it to modify its own code or data space; the kernel must then suspend the inferior, before eventually resuming execution of the debugger. This burden is relieved in UNIX versions such as AT&T System **V** Release 4 that support the "/proc" virtual filesystem [Kill84], because this allows a debugger to read and write the address space of its inferior process without having to undergo context switches.

Similarly, resuming execution of the inferior (e.g., by issuing the **cont** command) would necessitate multiple calls to *ptrace*() to re-insert all the breakpoints. This costly behavior would have had a chilling effect on the usage of Dalek's broadcast breakpoint facilities.

Dalek solved this problem by intelligent avoidance. We chose to remove and re-insert a breakpoint only when it was actually executed by the inferior. To circumvent the problem of framing errors in a disassembled listing, we utilized the data already stored in gdb's existing data structures regarding breakpoints. Should Dalek encounter a breakpoint instruction during disassembly of a section of the inferior's code, Dalek consults its gdb-inherited breakpoint data structures to access a copy of the machine instruction normally residing at that address. That copy can then be disassembled, furnishing an accurate display for the user without ever writing to the inferior's codespace.

5.4.7. Semantics of Stepping onto a Breakpoint

In the course of implementing Dalek, it must be admitted that Dalek itself had (at least) its own fair share of bugs. One bug in particular is interesting because it provides a classic example of how, when portions of a program are upgraded to reflect a better understanding of the original problem domain, previously unnoticed or acceptable behavior can become intolerable.

When single-stepping under gdb on some architectures (e.g., Sun 680x0 machines), if a breakpoint instruction is encountered, the breakpoint is not detected and its associated commands are not interpreted. That behavior may be reasonable, and perhaps even desirable, in gdb. However, it would be problematic, at best, in Dalek because the purpose of the breakpoint's associated commands might be to raise an event, or to correct a flaw in the inferior's logic and then **jump** around its incorrect code. Further reflection on the consequences reveals that such conduct on 680x0 machines would be totally unacceptable in Dalek since it would make free-floating events (i.e., events raised within **while—step** loops) incompatible with events bound to specific breakpoints.

In any case, the behavior of a debugger in such a situation should be similar across different machine architectures, in accordance with the Principle of Least Surprise. In contrast to gdb's inability to detect 680x0 breakpoints *at all* when stepping on them, not only will gdb *detect* a VAX breakpoint, but it will actually interpret a VAX breakpoint's commands just *before* stepping on it![4] Any implementation *ought* to ensure consistent semantics for its language, regardless of the underlying hardware.

The details of how Dalek's implementation ensures a uniform response to such a situation across different architectures are both gory and machine dependent, and will not be discussed here. Nevertheless, this issue provides a striking illustration of the importance of testing the operation of different features in conjunction to ensure that their *aggregate*

---

[4] This premature execution or "preignition" seems to be due, at least in part, to the nature of the decoding stage in the VAX CPU's instruction prefetch pipeline. Such behavior, although surprising, is only objectionable in a few cases. For example, suppose the user wishes to **step** right up *to* a breakpoint, and then poke around before proceeding further. The breakpoint's commands, should they contain a **cont**, will instead resume the inferior's execution, denying him the opportunity to inspect the inferior before that particular state is altered.

behavior is truly consistent with the user's expectations. While Dalek's behavior, unlike gdb's, is at least very similar on different hardware, we cannot claim its semantics to be absolutely *identical* on 680x0 machines and VAXen, since we have not yet bothered to compensate for the "preignition" problem on VAXen.

5.4.8. Garbage Collection

The routines in gdb that evaluate expressions were written to be as generic as possible. To allow a clean interface among these routines, gdb needed a single internal data structure capable of representing the value of *any* data object in the inferior process. Therefore gdb's designers defined the "struct value" as a universal way to represent the values of an inferior process's objects within the debugger's address space. All gdb's routines involved in expression evaluation take struct values as parameters and also return an object of that same type.

Several of these struct values are typically required during the course of evaluating even a moderately complex expression. Clearly the memory for such objects must be allocated dynamically, as needed. As a concrete example, the gdb routine value_binop (arg1, arg2, op) performs the binary operation denoted by "op" on its two other arguments, which may be any combination of integers or floating point numbers. Naturally its two operands, as well as its return value, are of type struct value.

Rather than making value_binop() responsible for freeing the storage of its two incoming operands once it has finished computing its return value, gdb's designers felt it was cleaner, and safer, to place all such intermediate values on a "freeable" list. If the contents of a struct value may be needed later (e.g., when a new convenience variable is assigned) gdb removes it from the freeable list. Since an expression in gdb can never stretch across more than one command, all garbage generated as intermediate values during expression evaluation can be safely freed by execute_command() just before it begins to interpret the next command.

However, expression evaluation in Dalek is more complicated, because consistent behavior at the user interface requires that expression evaluation be integrated with some of Dalek's other new features. Consider the following Dalek command:

set $x = **$udf1**() + **$udf2**()

In the course of evaluating the right-hand side of this expression, Dalek will invoke two user-defined functions and must therefore interpret commands in each of their respective bodies. Clearly, to adopt gdb's garbage collection policy — i.e., freeing the result returned by the first function call just before interpreting the first command in the body of the second function — would not be a safe practice.

Instead, Dalek's implementation must delay freeing an object until the command that created that object has been exited with no possibility of resumption. In essence, Dalek treats the freeable list as a stack. This has also been integrated with the error handling so that even if a command aborts, all the garbage it has generated will be cleaned up before the next command is interpreted.

5.4.9. Meta-debugging

In Section 5.4.5 on error handling, we alluded to the advantages of knowing the current dynamic context of the debugger (sometimes as important as knowing the inferior process's current context). A pathological example (Figures 4-4 through 4-6) in Section 4.2 illustrated how a user who fails to "keep it simple" could mire himself in a debugging quagmire. Although Dalek's meta-debugging features will not prevent such situations, they will allow the user to better appreciate the exact circumstances of his predicament in excruciating detail. Armed with this knowledge, the user will at least have a fighting chance of extricating himself, rather than being forced to admit defeat and to rerun the inferior from the beginning.

Because so much effort went into making the error-handling compatible with the meta-debugging features, the latter were relatively easy to implement. Unfortunately, to remain somewhat compatible with gdb (thus allowing Dalek to more readily be merged with a future version of gdb), certain compromises had to be made. One in particular detracted from the elegance of the meta-debugging implementation. Gdb's workhorse is a procedure named "execute_command()" which takes as a parameter either a line of text entered interactively by the user or the *text* field of an embedded struct command_line. Since execute_command() does not have access to all the fields of a struct command_line, it has no way of directly ascertaining whether the current command_line has its meta_break flag enabled. And although each of Dalek's "driving" routines uses a single global, "breakpoint_commands", to traverse the singly-linked list of struct command_lines that each is interpreting, by the time a driver calls execute_command(), it will have already shifted breakpoint_commands to point to the next pending command_line it should interpret. Therefore we were forced to maintain a global variable, "meta_break", in order to communicate meta-breakpoint information to execute_command() from "driving" routines like do_event(), do_breakpoint_commands(), do_while(), do_if(), and call_local_function().

The ramifications of the **edit** command also required some work. For example, suppose Dalek is interpreting a command block containing a **step** command that resumes the inferior. If the inferior then steps on a breakpoint, Dalek will need to "remember" its position in the first command block before interpreting any new breakpoint commands. It accomplishes this by stacking a pointer to the next pending command_line in the old command block; the remainder of that command block will be accessible via the command_lines' "next" fields. We will refer to such a stacked pointer as a *header*, although the reader should bear in mind that it will generally point not to the *first* command_line in a suspended block, but rather to the next command_line Dalek should interpret, once Dalek resumes that block.

Now if the new breakpoint's commands contain a **pause** or a meta-breakpoint, or if the user is in **dstep** mode, then the user will have the opportunity (once paused) to **edit** commands in a command block that is stacked "underneath" him. Since the user may delete a command_line from such an underlying command block, the code implementing **edit** must scan through the stack of pending headers in order to determine whether any of them points to this newly deleted command_line. If so, then in addition to "splicing" the newly deleted command_line out of its own singly-linked list, the code implementing **edit** must also update the appropriate header(s). Similar considerations will apply when the user inserts a new command_line in a block (although an insert capability has not yet been implemented).

Fully integrating the **edit** command into the debugging environment necessitated some new data structures as well. It was easy to get **edit** to operate on *named* command blocks, i.e., those command blocks with a fair degree of longevity such as breakpoint or event command blocks, user-defined commands, or user-defined functions. For example, the commands for breakpoint[5] number N are accessible by issuing the command:

edit N

The situation was not so straightforward with *nameless* command blocks such as a **while—step** loop typed by the user at Dalek's interactive level, or a lengthy **if** statement that Dalek was currently reading from a **source** file and interpreting. It might be argued that precisely because such blocks *are* nameless, there is no way syntactically for the user to specify any of them as a parameter to the **edit** command. But if the user had tediously entered a **while** loop from Dalek's top level, and then hit the interrupt key (e.g., Control-C) to effect a pause, he would be justifiably frustrated if he were unable to **edit** one line in that loop to correct a simple typographical error. Yet what syntax could possibly allow him to specify that it is this **while** loop that he wishes to edit? We felt it was imperative for the user to have the ability to **edit** such nameless blocks, so the obvious solution was to define a default semantics: the **edit** command, with no parameter, will access the *most recent* command block that has been stacked pending its resumption. In addition to solving the problem of access to a nameless command block, this default semantics has proven to be a convenient feature for accessing a named block as well, if such is indeed the most recent pending block.

Now because command blocks are implemented as *singly*-linked lists, we needed to

---

[5] If the user has also associated a command block with *signal* number N, then **edit** will query the user to resolve this ambiguity.

maintain not only a *header* pointing to the next command_line pending in each nameless block, but also a pointer to the *initial* command_line in each. To understand why this was necessary, consider a freestanding **while** loop: if the user is paused part way through the loop, the header will allow **edit** to access all *subsequent* command_lines in that block. Yet what about those command_lines near the top of the loop that precede the one indicated by the header? To permit editing access to *every* line in a nameless block, and because *multiple* nameless blocks might be suspended at some point[6] we had to maintain yet another global stack, "dedit_lambda_stack", to hold pointers to the initial command_lines of all pending nameless blocks.

5.4.10. Predefined Debugger-Language Functions

As a useful adjunct, we also implemented a number of predefined debugger functions. In contrast to user-defined debugger functions whose command blocks are interpreted by Dalek whenever they are called, the predefined functions are actually realized by compiled C language code. Since they are linked as part of Dalek's runtime image, they can exploit information and capabilities embedded in Dalek itself that may either be inaccessible to the user directly, or may be available only in a format that is inconvenient for programmatic use.

To promote a conceptually uniform interface between Dalek and the user, calls to these predefined debugger functions were made to appear syntactically indistinguishable from calls to user-defined debugger functions.

---

[6] For example, the user could type in a freestanding **while** loop, then hit Control-C, then **source** a file containing an **if** statement that itself **pause**d for interactive approval before resuming.

Among the first such predefined debugger functions we implemented was **$func**(N), which returns (as a string) the name of the procedure N levels above the procedure in the inferior's stack that has been selected as the current focus of attention. Although the user can easily glean this knowledge visually by scanning through gdb's response to a **back-trace** command, that interactive method offers absolutely no hooks for programmatic access to the same information.

Another extremely useful predefined debugger function is **$quiet_finish**(). Invoking this will cause execution of the inferior process to resume until the currently selected procedure returns, whereupon **$quiet_finish**() will then return — as its own value — the value returned by that inferior procedure. Normally, the procedure that is currently executing is considered to be "selected"; although the user may override this by explicitly selecting a procedure higher up in the inferior's runtime stack.

$quiet_finish() was relatively easy to construct, because we were able to employ much of gdb's existing code that implemented its **finish** command. Again however, merely extending gdb's functionality without forging a corresponding extension to its semantics resulted in a mechanism that was not sufficiently robust for its intended purpose.

In this case, a problem could arise if, e.g., the user issued a command such as

> **event set-attribute** ‘addr **$quiet_finish**()

Dalek, inheriting gdb's method, implemented **$quiet_finish**() by setting a temporary breakpoint at the return address of the currently selected inferior function.[7] The inferior

---

[7] Gdb was already able to restrict such breakpoints to the currently selected frame, so that, e.g., if the user issued a single **finish** command when the inferior was executing within a recursive procedure, execution would continue until that particular invocation of the recursive procedure returned.

was effectively then allowed to **cont**. If all worked as planned, the next breakpoint executed by the inferior would be that temporary one, whereupon the C code for **$quiet_finish**() would access the inferior function's return value.

However, if the inferior executed a *different* breakpoint before encountering the temporary one, gdb's code would remove the temporary breakpoint, unilaterally cancelling its mission without even notifying the user! Needless to say, importing such behavior into Dalek would have wreaked havoc with the accuracy of an event's attributes. It would have precluded the use of **$quiet_finish**() in breakpoint (or event) commands activated by *every* invocation of a recursive inferior function, or by any two inferior functions, one of which might (even indirectly) invoke the other.

The solution was for Dalek's code implementing **$quiet_finish**() to include a "driving" loop — essentially transforming the semantics to match the user's *goal-driven* expectations. Thus, for example, it is now possible to get meaningful results from debugging code such as Figure 5-5, where the user sets a breakpoint in a recursive inferior function, and includes some command that invokes **$quiet_finish**(). When the inferior process calls *factorial*(), this will result in the event history containing one occurrence of event ‘factorial (with the appropriate return value as its attribute) for every (possibly recursive) invocation of *factorial*(). Of course, if the user does *not* desire to record recursive calls, a command block such as that containing the **event raise** command in Figure 5-5 can be guarded by a generic test, i.e., "**if** (**$func**(0) != **$func**(1))". If the user wishes to detect *indirect* recursion as well, he must employ a loop and examine the names of every caller from **$func**(1) to **$func**(**$t_frame**()), much as in Figure 4-11.

In the future, Dalek could easily be enhanced so that it could access files such as /usr/lib/lint/llib-lc, to learn the return types of library functions in much the same way that the UNIX *lint* utility gains that information.

---

event define ‘factorial (‘ret_val)
end

break *factorial*
commands
  silent
  event raise ‘factorial (**$quiet_finish**())
  cont # Continue the inferior process.
end

Fig. 5-5 Use of **$quiet_finish**() in a recursive function.
