CHAPTER 7

Discussion

This chapter treats an assortment of topics. Although a fair amount of discussion has been interspersed throughout previous chapters, to have discussed other matters "inline" might have proved too much of a digression.

We begin by offering a more extensive comparison of Dalek's events, and also of certain of its semantic aspects, with those of some kindred debuggers covered in Chapter 1. After a brief respite in the form of an amusing (but true) story to motivate the subsequent section, we close with some reflections on the possible role events might play, when viewed from the wider perspective of the software life cycle.

7.1. Comparison of Dalek with Related Debuggers

Now that the reader has a better idea of Dalek's capabilities, and some sense of the new methodology it supports, we can revisit in more depth a subject that has cropped up several places in earlier sections — Dalek's relation to other debuggers.

7.1.1. Comparison of 'Events'

The term "event" is not terribly well-defined, even if we limit its sense to that used in debugging. Historically earlier debuggers, such as RAIDE and SNODDT, used "event" to denote essentially what, in Dalek, are referred to as "primitive events". Even so, the notions are still not quite equivalent. For RAIDE and SNODDT (as for Dalek), the notion of a (primitive) event was already overloaded — on the one hand, it refers to an action initiated by the *inferior* process, whereas on the other hand, it refers to the corresponding action taken by the *debugger* (which, for all three debuggers, was to interpret an associated block of debugger language code).

Even when limited to the first sense, "(primitive) event" is still overloaded — since an inferior’s "action" can mean either *one instance* of a particular action (e.g., the second time the variable *foobar* is written), *all instances* of that particular action, or a generic *class* of actions (e.g., writing into *any* variable).

If a few meanings are good, then more must be better. Therefore Dalek adds to this plethora a third, entirely different sense of "event". Whereas in both historical senses, events were viewed as *actions*, i.e., as procedural entities, Dalek insists that events also have a non-procedural aspect which it represents in the form of *data*. Thus Dalek maintains a permanent record of the attribute values of every event instance.

In Section 3.4, we suggested that Dalek’s browse mode for *visually* reviewing the event history could assist the user in generating new debugging hypotheses. The example in Section 3.4.1 demonstrated that *programmatic* access to the event history can also be quite beneficial. The key point to storing event occurrences as data is that it permits "raw" data to be *re-interpreted* in the light of subsequent debugging hypotheses or semantic frameworks, through either visual or programmatic access modes. Thus Dalek provides support for the user’s mental model to evolve so that it more adequately represents the causal structures that underlie the inferior’s observed behavior.

In terms of our performance metric for two-process debugging, any such re-interpretations are essentially "free", i.e., no context switches are required. Given the fact that the inferior may not have manifested certain symptoms until after it had been active for several hours in realtime, this "free" cost of subsequent re-interpretations is not only a convenient measure of machine resources, but also is a realistic indicator of certain finite human resources — e.g., time, patience, and alertness.

Another important difference between events, as supported by Dalek, and events in virtually all other debuggers, is that Dalek provides a far more powerful means of specifying and recognizing events. The dataflow graph is certainly convenient and conceptually clean for users to work with, but the real power comes from the nature of the code that can reside at any high-level node in that graph. We note that semantic interpretation functions akin to those rigorously formalized by Tarski [Tars35] seem to be powerful enough to map from the syntax of any formal language into arbitrary "meanings" in the world.[1] Although such functions are typically rather straightforward mappings between individual symbols (or at least contiguous substrings of symbols) and the entities, classes, or properties they represent, such need not necessarily be the case. In Dalek, we allow any semantic interpretation function residing at a high-level event to be an *arbitrary* Turing computable function. We do not limit event recognition to just those semantic interpretation functions that can be computed by regular expressions or by pushdown automata. Rather, each high-level event has full Turing capability, should the user need it. The inability of certain other methods to recognize non-contiguous sequences of symbols in the event stream or non-regular expressions places an important practical constraint on their ability to support arbitrary, user-defined abstractions.

---

[1] Indeed, [Tars35], along with the early work of Carnap [Carn47], comprises not only the foundation but also several of the lower floors in the edifice currently known as denotational semantics.

7.1.2. Other Semantic Comparisons

It is interesting to speculate that the divergent evolution of Dalek and RAIDE may, in large part, be due to the fact that certain features of RAIDE's language were never actually implemented. For example, the RAIDE language definition includes two very similar constructs:

> **while** (condition) **do** {commands}, and
> **when** (condition) {commands}.

The former is merely the familiar loop construct, whereas the latter specifies the recognition criteria for an event and the commands to be executed *whenever* the inferior's state satisfies those conditions. Thus the latter construct is synonymous with a Dalek "free-floating" event, although its syntactic sugar perhaps makes it appear more palatable. Since neither construct was ever implemented, RAIDE's designer may not have been aware that his construct for free-floating events could have been implemented in terms of the **while** statement. He may also have failed to appreciate the potential for semantic tension between the two constructs.

Johnson noted regarding RAIDE, "The most serious shortcoming of the work . . . is that it is not user tested". Perhaps if it had been used extensively, even if only by its designer, it might have been noticed that even seemingly innocuous command sequences like "**execute** 3 statements; display *foo*" can have problematic semantics if their single-stepping of the inferior triggers an event or causes it to encounter another breakpoint. In such a case, would the first command block have been stacked so that, upon its resumption, the value of *foo* would eventually have been displayed? Although we may never know for sure, indications are that RAIDE was not designed to support stacking of partially-interpreted command blocks.

As evidence, RAIDE contained a command to **break**, meaning *not* that a breakpoint should be set, but rather that control should return to interactive mode. Regarding its semantics, the RAIDE language definition explicitly states that when a **break** command is interpreted, "the deferred action (*i.e., the event command block*) or debugging procedure containing the **break** is terminated". In contrast, when Dalek's **pause** command is interpreted, any subsequent commands in its block are stacked, so that interpretation of that command block can later be resumed. The implementor of RAIDE seems not to have considered the possible semantics of stacking command blocks in cases where the flow of control had "tunneled through" the inferior. The quintessential example of such tunneling, as covered in Chapter 6, would be where a **while—step** loop causes the inferior to execute a breakpoint that has an associated debugging command block.

The SNODDT debugger supported the equivalent of Dalek's **pause** command and allowed partially-interpreted command blocks to be stacked [Hans90]. Thus in certain respects its semantics were fairly similar to those of Dalek. However, because SNOBOL4 ran on a *virtual* machine, there is an important distinction that must be made: SNODDT could "watch" a variable merely by notifying the virtual machine, whereas Dalek — since it must debug compiled code — must perform the virtual machine's job itself, typically by employing some form of **while—step** loop. Given an infinite number of hardware watchpoint registers, Dalek and SNODDT would essentially be on an even footing: Dalek would no longer need to employ imperatively-programmed loops to watch variables, and so even the simpler forms of such loops would not be susceptible to false negatives[2] should such a loop cause the inferior to encounter a breakpoint in its code. Nevertheless, both Dalek and SNODDT would still need to exercise caution to avoid false positives, since, e.g., if a debugging routine were to alter the value of a watched inferior variable, control would immediately be transferred to a command block that might try to record that as an event occurrence.

7.2. True Confessions

For two very different reasons, it may be instructive to relate an experience we had during the development of Dalek's broadcast breakpoint facility. When Dalek's ancestor, gdb, would read in an inferior's symbol table, it maintained a separate data structure, "misc_function_vector", to hold the names and addresses of library routines. After implementing and testing Dalek's new mode to broadcast breakpoints to all the *user's* procedures whose names matched a given pattern, we felt the user should be given the opportunity to broadcast to library routines as well. It seemed this capability would be relatively easy to implement, because gdb had already loaded "misc_function_vector" with all the necessary information. Cursory testing was then performed on the new capability, and everything seemed to work properly. For example, given a pattern such as "p*", breakpoints were properly set at library routines like *printf*().

A few days later, in order to perform a complicated action efficiently (measuring conditional probabilities for the inferior's runtime stack behavior, as shown in Section 4.5, i.e., Figure 4-12), we broadcast a set of breakpoints to *all* procedures, including the library

---

[2] Recall that the TOPDOG construction is an imperative watch loop that is *not* susceptible to false negatives.

routines. But when the inferior was executed after being instrumented with this debugging code, the inferior would abort mysteriously due to segmentation faults inside *malloc*(). Even if we selectively **delete**d *every* breakpoint that the inferior actually encountered before its abnormal termination, it would still segfault. This was quite perplexing, because the inferior would terminate normally when it was *not* being watched by the debugger! In spite of the fact that the debugger and the inferior were running in two disjoint address spaces, somehow the mere "observation" of the inferior process by the debugger was radically altering the course of its execution. It was not at all clear what had caused this sudden change of behavior. After several hours of debugging, the culprit was finally identified: When the code Dalek inherited from gdb was reading the inferior's symbol table, not only did it load the names of all library *functions* into "misc_function_vector", but it also loaded that data structure with the names and addresses of the library's *data* symbols, such as *errno*!

The cause of Dalek's new bug was quite simple, although extremely difficult to detect: we had been setting breakpoints in the inferior's *data segment*, thus overwriting valid data! Of course, if the inferior had ever tried to *execute* such a word of data, Dalek, after breaking, would have obligingly replaced that word with its original contents. However, Dalek had no way of knowing that the inferior was trying to *read* words that had been overwritten with breakpoint instructions. This problem was extremely difficult to detect because it happened in an early phase of Dalek's development before we had optimized the breakpoint insertion algorithm. So at this stage, whenever the inferior stopped and control was returned to Dalek, Dalek's very first action (even before executing any breakpoint commands or resuming interactive mode) would be to remove all breakpoints;

as soon as any command was issued to resume the inferior, Dalek would re-insert all those breakpoints again. Thus, whenever *we* regained control after single-stepping at the machine instruction level, and then inspected various memory locations, everything inside *malloc*() seemed to be fine. Yet, during each tiny step, whenever the *inferior* looked at one particular data location, it contained a breakpoint instruction!

If any lessons can be drawn from this incident, one would have to be that, with the current (lack of) operating system support for debugging, the user must be on guard against any unintentional interactions between the debugger and the inferior. Even in systems like UNIX, where the two run as separate processes in disjoint address spaces, there is no guarantee that monitoring the inferior process will not disturb its "normal" behavior pattern if breakpoints are employed.

Even if we refrain from indiscretions like setting breakpoints in the inferior's data segment, it is still possible that certain inferior programs were designed to *read* portions of their own code, and treat them as data. If such is the case, then even setting a breakpoint in the inferior's code segment has the potential to alter its behavior. Similarly, if an inferior purposely loads code in its data segment and then begins executing there, how can the debugger know exactly which parts of the data segment are safe to instrument with breakpoints? It could be argued that compiled C programs have no business treating their own code as data, nor vice versa, but without hardware protection to prevent this sort of activity (or at least to announce its occurrence), the objection is moot.

Another important point (and the primary reason we include this embarrassing story) is that many bugs are fundamentally *semantic* in nature, and no amount of, e.g., runtime array-bounds checking, can reveal their presence. Specifically, such checking would be unable to detect the semantic flaw that a structure named "misc_*function*_vector" actually contained the names of some objects that were not of type procedure. Thus even otherwise impressive products like Saber-C do not provide sufficiently powerful tools to solve the more general problems of debugging. For that matter, no event-based debugger based solely on *predefined* event definitions would be able to contend with such high-level, essentially semantic bugs.

7.3. Behavioral Debugging in the Context of the Software Life Cycle

7.3.1. An Event for All Seasons

Along with more traditional structured software-engineering methods and formal verification, events can provide an important technique for managing several stages in the software life cycle. For example, when prototyping is employed in order to obtain a more realistic "requirements analysis" from users, events could be used to ensure that the specific behaviors desired by the users are preserved between the prototype and the early production versions.

Events appear to be exceptionally applicable to the evolutionary phases of the software life cycle, especially those awkward stages before a program has matured and become relatively stable. The stimulus for a program to evolve is frequently either a desire on the part of its user community for enhanced semantic functionality or a change in the semantics of the environment in which it is used. Thus, tools and methodology for *semantic debugging* seem particularly applicable during this phase. Of course, the use of events should not be confined to the evolutionary phase of a program. On the contrary, the best time to build up a customized library of events is during the initial *development* phase. As mentioned, such events could prove beneficial not only for debugging during prototyping, but also in the transition from prototype to production status. Perhaps more importantly, those individuals actually involved in writing a piece of software are uniquely qualified both to define events covering its major semantic behavior, and also to see that those events are bound to appropriate locations in the source code.

For example, calls to *malloc*(), the memory allocator, might occur at many different places in the source code of a large, complex program. It would be simple to put a single breakpoint in *malloc*() and raise an event there. Yet the vast majority of the dynamically allocated objects might never be freed later. If the purpose is to debug a garbage collection module, then many of the ‘malloc events would be superfluous. It would be more appropriate to put breakpoints in exactly those routines that call *malloc*() to allocate space for just those types of objects that will later be subject to garbage collection.

It would be a relatively easy task for a program’s developers to establish a customized library of events for it, and the completeness and quality of the event library they produced would probably be far superior to the alternative — several years later, after the developers have moved on, a new group of employees could be hired to support an unfamiliar program.

Events could also serve as an important form of documentation, linking *syntax* (procedural code) with *semantics* (non-procedural specifications) [Solo87].

7.3.2. Enhancements for Binding Primitive Events

Bound events could be made "adhesive", i.e., bound to "semantically fixed" points in the source code by use of *lint*-like directives embedded in the source code. Since such directives would appear as comments to the C compiler, their presence would not alter the inferior's object code. As stated previously in Section 0.2, this is an important consideration.

Thus, rather than a **source** file of Dalek commands, e.g., calling for a breakpoint to be set at line 55 of main.c, and specifying, among its commands, one to raise a particular event, the user might instead invoke a new Dalek predefined function, perhaps named **$scan_line**(). Given a filename and a pattern as arguments, **$scan_line**() could broadcast breakpoints to every line of source code in that file that matched the given pattern. Typically, the pattern would be one of the *lint*-like directives, embedded in the source file for exactly that purpose. But such a capability might also offer a more efficient method for watching a particular variable, although only *textual* references would be caught, and not accesses to the variable via pointers.

For that matter, we could implement another new predefined function, **$bind_commands**(), which, given a filename, would always search for the *lint*-like directive denoting "break at this line". Once at such a line, **$bind_commands**() would expect to read in a customized command block to associate with that particular breakpoint. Thus we can extend Knuth's notion of WEB, [Knut84] — in which the code and the documentation are intermingled in the same file — to one in which the means of specification and verification are embedded as well! Note that a variant of **$bind_commands**() could search for appearances of the "assert" macro in the source code. When it encountered an assertion, this variant **$bind_commands**() would set a breakpoint and construct its command block from the C language text of the assertion. Thus, assertions that are only *conditionally compiled* into the object code could nevertheless be dynamically monitored by Dalek, even on a "stripped down", production version of the inferior, to which Dalek attached after it was already running.

7.3.3. Incremental Event Coverage and 'Lite Validation'

The idea behind a normal "validation suite" is to run an inferior program through a series of tests, and then to compare all its output with a "standard", typically the output from a previous, trusted version of the inferior program run on the same set of test data. All output would be redirected into a file so that later, the UNIX *diff* utility could detect any differences.

We propose a technique, called "lite validation", that would seem to have two distinct advantages over the traditional use of a validation suite. As we envisage the new method, the user would instrument the inferior with a variety of events, in an attempt to more or less "cover" whatever behavioral characteristics were deemed to be semantically important. Thus, rather than observing only the effects of the inferior on its external environment, we would also be monitoring its internal workings.

One consequence is that we could collect far more data, some of which might indicate internal flaws that *contingently* happened not to be reflected in its output given *that* particular of test data. These would thus be latent bugs, waiting to bite when "awakened" by the proper set of input data.

Since conventional validation runs in batch mode, it can only detect an error long after the inferior has terminated. In contrast, because "lite validation" would monitor behavior at runtime, it could suspend the inferior as soon as any event or attribute were reported that deviated from the intended behavioral specification. The advantage here is that we would presumably be very "near" to the point of infraction. That would make it much easier for us to isolate the cause of the bug — rather than merely noting, perhaps hours later, that its symptoms had manifested in the external environment by leaving their "tracks" in the output logging file.