CHAPTER 1

Flora and Fauna in the Genus 'Debuggerae'

In this chapter, we attempt to position Dalek and its features in relation to other debuggers and programming environments. Naturally Dalek was not conceived in complete isolation from previous work in computer science. In retrospect, it appears to have a number of fairly reputable conceptual progenitors. While this appearance of a direct evolutionary descent may prove helpful in evaluating the significance of our work, Dalek was in fact *not* intellectually descended from notable historical debuggers such as Magpie, or those for the Lisp or SNOBOL4 languages.[1] Rather, Dalek's creation was a response to our frustration with the inadequacies of existing UNIX tools such as adb and dbx that came bundled in the standard UNIX distribution.

1.1. Debuggers Are Inherently Machine-Dependent

The nature and capabilities of debuggers have always been strongly dependent on the operating systems and underlying architectures (or in some cases, the virtual machines) under which they and their "inferiors" run. As an obvious example, since it is impossible to set breakpoints in ROM, it is rather frustrating for a PC user, aided only by the ability of the CPU to single-step at the instruction level once its "trap" flag is set, to observe an inferior executing a BIOS (Basic Input Output System) routine under DOS. An even

[1] In fact, when we began work on Dalek, we were relatively unfamiliar with many debuggers operating in interpretive or incrementally compiled environments. Our knowledge of debugging tools targeted to run under non-UNIX operating systems was minimal. For example, we independently invented "full programmability" of the debugger's language, and only later discovered that some similar features were included in the newest version of VAX DEBUG running under VMS, to which we did not have unimpeded access.

more common constraint on debuggers is that there is seldom hardware support for monitoring the bus to determine when particular memory locations are accessed. This ability can be extremely helpful when the user desires to watch a variable or array to discover exactly when or why its contents are accessed or altered.[2] Another constraint for debuggers under UNIX is that the debugger and the inferior are (generally) two different processes with different address spaces. This can make transfers of control between the two expensive, since it involves at least one context switch by the operating system. On the brighter side, this separation of powers ensures that the inferior cannot clobber the debugger’s code or data space (unlike debugging on PC’s under DOS).

Some designers of debuggers have apparently concluded that if you cannot have the desired support for debugging in hardware, it is worthwhile to construct a virtual machine (sometimes specifically to aid debugging) under which the "inferior" can run. The price of such enhanced debugging capabilities is necessarily a loss of speed, since an inferior executing on top of an extra layer of software simply cannot run as efficiently as one composed entirely of native machine code. Clearly though, if (for whatever reasons) a decision has already been made to operate in an interpreted environment, then excellent debugging support can usually be provided while incurring only a negligible additional penalty in running speed.

[2] The Intel 80386 chip is the first microprocessor to offer this capability, although having only four "data watchpoint" registers still severely limits the number of memory locations that can be watched simultaneously.

1.2. Debuggers with Limited Events or Programmability

1.2.1. RAIDE

RAIDE (*R*untime *A*nalysis and *I*nteractive *D*ebugging *E*nvironment) [John79] demonstrated (among other things) the feasibility of a language-independent debugging environment. RAIDE required the construction of language-specific "compilers" to generate what can best be described as intermediate code to run on a virtual machine named SPAM (*S*pecialized *P*ro-debugging *A*bstract *M*achine). In addition to the barebones SPAM code, these compilers also emitted code to perform extensive runtime error checks (e.g., to detect array bounds violations).

The primary purpose of SPAM was "to provide the user's program with a machine environment that is not hostile to the debugging process". SPAM incorporated a tagged architecture, wherein every object (constant, variable, or procedure) had a tag pointing into the symbol table. This permitted runtime type-checking of actual versus formal parameters, and of the values assigned to variables. Attempts to reference uninitialized variables could be similarly caught through this mechanism. In addition to type descriptors, entries in the symbol table also contained several "interrupt" fields. If enabled, such fields informed the interpreter that the inferior program's execution on SPAM should be interrupted, e.g., whenever a particular variable was accessed or updated.

The luxury of having not only a virtual machine, but also compilers that were created expressly to support debugging, allowed a powerful event-action language to be designed.[3] The symbol table's "interrupt" fields provided convenient hooks whereby

---

[3] Unfortunately, the language design provided no means for the user to define *hierarchical* events. In addition, two constructs that, in principle, allowed more complex primitive events to be specified were never

primitive events such as the reading or writing of a variable, or the entry into or exit from a procedure could be recognized. Such events could be associated with arbitrary actions expressed by the user in a fully programmable language.

It is interesting that, on the surface, the semantics of two of Dalek's most significant features — events and full programmability — appear quite similar to the semantics of their counterparts in RAIDE, even though we only learned of the extent of Johnson's work long after those features were implemented in Dalek. Since RAIDE was not available under UNIX, and since no analogous tool existed for compiled inferiors, we were driven by necessity to invent a debugger with some similar capabilities.

1.2.2. SNODDT and the SNOBOL4 Programming Environment

Although the SNOBOL4 environment offered event and action capabilities [Hans78] quite similar to those of RAIDE, the intention of its designers was quite different. Their purpose was to construct an *integrated* environment in which the language of the inferior could also be used for debugging. A package of special-purpose debugging routines called SNODDT was implemented. When "compiled" with the inferior routines, SNODDT's commands permitted the user to express concepts such as single-stepping, which did not appear explicitly in the SNOBOL4 language. As in the RAIDE system, SNOBOL4 source code was first "compiled" into intermediate code, which was then interpreted for a virtual machine. Primitive events like the reading or writing of a variable could be recognized at essentially no extra cost beyond that already incurred by the interpretation scheme. In SNOBOL4's environment, the "code" for the debugging

---

implemented due to their "ramifications". This point is elaborated more fully in Section 7.1.2.

actions resided in the same address space as that for the inferior, but because of the nature of the SNOBOL4 virtual machine, it was not susceptible to corruption by the inferior; rather, both the inferior and the debugging code were equally susceptible to corruption by the interpreter itself (presumably an extremely unlikely occurrence).

SNOBOL4 had a fixed set of events, essentially the same set as in RAIDE. For example, in SNOBOL4 it was not possible for the user directly to express an event such as "$x < y$". Instead, an event like this could be kludged together by associating code with *two* events that *could* be expressed in SNOBOL4, namely updates to variables x or y. As in RAIDE, aggregated or hierarchical events were not supported.

1.2.3. Magpie

The Magpie [Deli84] environment was somewhat similar to that of SNODDT in that its debugging language was a superset of the language used in writing the inferior (Pascal). However, an extremely significant difference was that Magpie did *not* utilize an interpreter or a virtual machine. Instead, debugging actions were realized through incremental compilation. Since the inferior's "code" under Magpie really was native machine code, there was no interpreter to trap all references to a given variable by a non-existent virtual machine. Instead, events such as the reading or writing of a particular variable were recognized *by inserting extra machine instructions in the inferior's object code* wherever that variable appeared *textually* in the inferior's source code. Magpie already had to maintain a parse tree for the inferior in order to support its "normal" mode of incremental compilation in case, while suspended at a breakpoint, the user chose to modify sections of the inferior's source before allowing its execution to resume. Given the prior existence of this parse tree, it was fairly easy for Magpie to maintain a database that (among other things) kept track of where every identifier appeared textually in the source code. Thus if the user decided that all writes to a particular variable should be associated with some action ("demon"), Magpie could augment the inferior's object code, instrumenting all writes to the monitored variable with machine code instructions to call the corresponding action routine. Due to the limitations of a native machine code run-time environment, Magpie's demons were incapable of noticing when the contents of a monitored identifier were altered via an alias or pointer. Thus some of Magpie's event recognition was not immune to "false negatives"; i.e., some monitored events could occur, yet fail to be recognized.

Like SNODDT, Magpie allowed statements to be executed in the context of any activation record on the stack, including statements that called inferior functions. Again though, due to the fact that the native machine code for newly compiled debugging routines was loaded into the inferior's address space, the user's debugging code and data were susceptible to corruption by the inferior.

Although Magpie's original implementation only allowed new *local* variables to be declared during debugging, its authors were acutely aware that the ability to declare new global variables would be extremely useful to support debugging. Fortunately, Magpie's implementation could have been modified to allow the declaration of new global debugging variables without violating the spirit of the paradigm Magpie presented to the user.

1.2.4. Cedar

The goals of the Cedar [Teit85] project included the creation of a highly integrated environment operating in what its designers termed an "open" system. By this, they meant that multiple processes (including every component of the operating system itself) would all execute in a large, shared virtual address space. Clearly, the nature of debugging in such an environment was quite different from that under UNIX, since not only was a debugger generally susceptible to corruption by an inferior, but also the inferior's data was not protected from intentional or accidental alteration by other processes.[4] Because even an application program had the ability to reboot or power-down the machine on which it was running, over-the-ethernet debugging was commonly employed.

Although debugging in Cedar could, on the whole, perhaps more appropriately be compared with kernel debugging under UNIX, one particular feature is relevant to the present discussion. The Cedar language, like the C language, was ordinarily compiled into native machine code. However, an interpreter for Cedar source code was also available in the environment. Thus a user could "attach" to an inferior process and suspend its execution, then type arbitrary Cedar statements or expressions. These would then be evaluated in the context of the inferior process. The open literature on Cedar [Dona85] [Swee85] [Swin85] makes no mention of any notion corresponding to events.

---

[4] Apparently a "Core Wars" situation was avoided because most applications were written in a "safe subset" of the Cedar language that was guaranteed to prevent harm to underlying storage systems such as the virtual memory and garbage collection modules.

1.2.5. Lisp

Many dialects of Lisp, including Interlisp [Teit78] [Sand78] and FranzLisp [Wile84], support an **advise** construct for debugging. This construct provides an ability — analogous to that when debugging more conventional, imperative languages — to set breakpoints at procedure entries or exits, but it is somewhat more structured. The strategy underlying **advise** is that it permits the user to treat Lisp functions as black boxes, and only modify behavior at the *interfaces* between functions. Interlisp even allows the user to be advised only at a *particular* interface, e.g., where Routine_B is called by Routine_A.

Since the **advise** feature works not only for interpreted Lisp, but also for *compiled* Lisp expressions, part of its functionality may actually be achieved through the use of machine-language breakpoints. However, for interpreted code, **advise** can actually modify the function's source code to achieve the user's objectives, although it usually limits its changes to altering the function's property list. The user can specify arbitrary Lisp code to be executed at such "advise-points", e.g., to print a message, invoke a new read-eval-print dialogue, conditionally modify incoming parameters, or alter the value being returned by a function.

In addition to **advise**, Interlisp also offers a separate, less constraining debugging package whose specific additional capabilities allow the user to examine the dynamic call chain (programmatically as well as visually) or to "single-step" the Lisp interpreter through S-expressions. Clearly, many things are feasible in an interpretive environment that are impractical for compiled code. For example, in its debugging package, Interlisp supports a **breakin** command. Invoking **breakin** causes an editor automatically to modify the source code. Thus, not only does **breakin** allow the user to break at an arbitrary "line" (S-expression) within a function, it also allows him to break upon a particular evaluation of a variable.

Zetalisp [Fish90] followed this trend nearer to its logical conclusion, enabling the user to specify *patterns* for Lisp S-expressions. Whenever the interpreter evaluated an S-expression matching such a pattern, it could interrupt the normal flow of execution before, after, or during the evaluation, and invoke another read-eval-print routine to interact with the user.

Such a feature is, in principle, capable of subsuming not only both modes of Dalek's broadcast breakpoints, but also could presumably provide an inexpensive realization of monitoring every access to particular variables, or to Lisp atoms in general. In addition, *if* the proper hooks were available to the user (as they are in FranzLisp), then he could specify that, rather than having control revert to interactive mode when an evaluated S-expression matched a given pattern, control should instead be transferred to a user-defined routine. This would allow a user to implement his own event-recognition and handling mechanisms.

1.2.6. Saber-C

Saber-C [Kauf88] is a relatively new commercial product that runs under UNIX. In addition to a refreshingly user-friendly implementation of *lint*-style static checking of types and interfaces, it also provides an interpreter for C language constructs. This gives the user considerable flexibility to express debugging actions when the inferior process is suspended at a breakpoint.

The most unusual aspect of Saber-C is undoubtedly that it supports the execution of a *mixture* of compiled and interpreted C code. Interpreted code runs on a virtual machine with extensive debugging support (e.g., runtime array bounds checking, and dynamic verification that no data location is read before being written). Unfortunately Saber-C does not currently support events, and its ability to debug the compiled sections of an inferior's code mixture is apparently limited to dbx-style features. Although the user can adjust the mixture between compiled and interpreted code, this requires that the inferior's execution be restarted from the beginning.

Saber-C can trace *some* writes to an arbitrary memory location (even if that address was allocated dynamically), but only those references from interpreted code will be caught, not those from compiled code. To its credit, Saber-C does allow the user interactively to specify a block of code that can be associated with a breakpoint. Such blocks can also be called when Saber-C detects that a memory location being "watched" has been modified (by interpreted code).

Saber-C's interactive workspace acts much like another C file, allowing the user to define and manipulate auxiliary debugging variables, including new *global* variables.

Unlike most debuggers that are strictly for compiled languages, Saber-C and the "inferior" actually run as a single process. Because the Saber-C interpreter resides in the same address space as the inferior code, the interpreter is itself susceptible to corruption since a bug in a region of compiled code could cause the process to write to an arbitrary location indicated by an errant pointer. Thus the user should only compile code he considers to be extremely trustworthy, since by doing so, he waives not only all the runtime checks that are routinely applied to interpreted code, but also the protection they afford.

1.2.7. VAX DEBUG

VAX DEBUG [VMS88] provides almost the same degree of "full programmability" as does Dalek. It operates on inferiors written in languages such as C, Pascal, and Fortran that are compiled into native machine code. VAX DEBUG runs only under the VMS operating system, where it must first be linked with an inferior's executable file. Thus a single process executes both the inferior's code and the debugging services. These two semantically distinct sections of the inferior communicate through standard exception handling mechanisms. It is likely that at least the data, if not the code, of the debugging section can be corrupted by the inferior, but this is not clear from the available literature.

It must be admitted that although our idea of broadcasting breakpoints to the entries of all procedures whose names match a given regular expression was original (and unique to Dalek, as far as we know), the second broadcast mode — in which breakpoints can be written at every instance of a particular opcode — was implemented only *after* we learned of a similar capability in VAX DEBUG.

1.3. Standard UNIX Debuggers

1.3.1. dbx

Many variants of dbx [DBX83] exist. All operate on inferiors composed of native machine code that run as separate processes. The most powerful versions of dbx allow the user to associate a limited subset of their commands with particular breakpoints. No version of dbx has anything approaching full programmability. Only the most restricted notion of events is provided to aid the user through various forms of *tracing*. Tracing a procedure entails that the user will be notified when it is entered and exited. A variable may also be traced: Dbx will print any new values assigned to that variable, and may optionally suspend execution of the inferior as well. The most powerful versions of dbx allow certain expressions to be traced as well. For example, on a SUN3, the user can order dbx to execute a particular block of (limited) commands whenever the inferior's state satisfies a given expression. Although that expression can mention inferior variables, it cannot call inferior functions. No version of dbx allows the user to declare auxiliary debugging variables.

1.3.2. adb

Adb [Mara79] is, in many ways, an outmoded debugger. It was originally intended to operate only on machine code. Provisions were later made to incorporate certain high-level language features such as recognizing variables by their symbolic names. Although adb's syntax is a study in inscrutability, it is theoretically possibly to associate some of its commands with breakpoints. Its primary distinction is probably that it includes support for kernel debugging.

1.4. Operating System Tradeoffs: Integrity versus Efficiency

In classifying various debuggers, one can obviously use several criteria. Debuggers can be differentiated on the basis of their degrees of programmability or on the levels of support they provide for events. Certainly such categories help to illuminate the subject of debugging, but it is also instructive to classify debuggers along a third dimension, orthogonal to both of these others. We may inquire regarding not only what capabilities a debugger provides, but also regarding how the interaction with its inferior is structured.

Probably the most fundamental distinction along this "structural" axis cleaves debuggers into two camps based on whether or not the debugger and its inferior run as two separate processes under the operating system. As mentioned in Section 1.2.7, VAX DEBUG, running under VMS, is implemented as an exception handler. Thus a single process executes both the inferior code and also the debugging routines. In such a scheme, the debugging code and data cannot be protected from accidental corruption by errant pointers in the inferior.[5] Saber-C (Section 1.2.6), with its mixture of interpreted and compiled code, would be similarly susceptible if the errant pointer were written by compiled code. An additional concern regards memory management — if the debugging routines need to dynamically allocate memory (e.g., to hold newly written debugging commands, or newly declared debugging variables), then this would generally cause the address space to become interleaved, with one block of memory from the heap used by the inferior routines, and the next block used by the debugger. Suppose, in such a case, that the user decides to raise an event every time an inferior routine allocates memory from the heap, and to store, as that event's attribute, the address of the newly allocated block. Yet this attempt to record the inferior's behavior would alter the very phenomena the user wishes to observe! Thus, at least for compiled code, allowing a single process to execute

---

[5] In theory, if the debugging routines were the first to gain control, they could order the operating system to write-protect exactly those pages "belonging" to the debugger just before control transferred to the inferior. If the operating system were familiar with the tale of Ulysses and the Sirens, then it might be possible to instruct it to ignore any subsequent requests to remove that protection unless such requests were initiated from *within* that same protected space. That way, we could at least guarantee the safety of the debugger's code and data. Although we know of no operating system that supports such a capability, even if it were available, the "natural" layout of the inferior's address space would still be considerably altered, introducing the possibility of unknowingly disturbing the inferior's "natural" behavior.

both the inferior code as well as the debugging routines is at best problematic. With interpreted languages, this may not be such an important issue, since presumably most interpreters provide elaborate safeguards to ensure the integrity of the inferior's memory model.

But the price of integrity seems invariably to be a loss in efficiency. For interpreted code, the consequent execution time penalty is well known. For compiled code, when two distinct processes are involved in the debugging activity, transferring execution between them necessarily involves an operating system context switch. While this particular sort of context switch might conceivably receive especially favorable treatment from the operating system's scheduler, such does not appear to be the case in many versions of UNIX. Suppose, for example, the debugger wants to single-step the inferior. It would initiate that action by invoking the *ptrace*() system call. It would then call *wait*(), causing itself to be removed from the operating system's ready queue. Conceivably, in a worst-case scenario under a round robin scheduling regime, *every other process* in the system might be given a chance to run before the inferior was awakened and told to execute its next instruction. Only then would the debugger be placed back into the ready queue, but it would now be last in line behind all the other processes. Evidently, interpreting a **while—step** loop (an important construct in Dalek) could be an extremely time-consuming activity under such circumstances. Even on a lightly loaded system, the CPU time involved in context switching between the debugger and its inferior might comprise an appreciable fraction of the total charged to those processes. On a heavily loaded system, the passage of real time experienced by the user could deter him from employing the **while—step** construct in the future.

There may be a second source of inefficiency when performing debugging using two distinct processes, since this setup typically calls for the processes to operate in two disjoint address spaces. Clearly this prevents the inferior from corrupting the debugger's code or data, but how is the debugger to read variables from the inferior's data segment, and write breakpoints in its code segment? Under most versions of UNIX, including those for Sun and VAX machines, a debugger is allowed only extremely restricted access to its inferior's address space: it must use the *ptrace*() system call to read from or write to that address space. Once again, this requires a context switch. In fact, under VAX Ultrix, *each word* of data transferred between a debugger and an inferior requires two context switches, one to access the state of the inferior, and one to reinstate the debugger!

Fortunately, this second potential source of inefficiency is not an intrinsic characteristic of all UNIX systems. Versions such as AT&T System **V** Release 4 support a virtual filesystem named "/proc". [Kill84] Given suitable access permission, this allows a debugger to read or write an inferior's address space using the normal *read*() and *write*() system calls, just as though the inferior process were an ordinary file. This feature considerably widens the bandwidth available for communication between a debugger and its inferior process, since no context switch is required unless execution of the inferior is actually resumed.