CHAPTER 0

Introduction

Although the volume of published research on debugging sequential, compiled programs has been minimal since the late 1970's, a countervailing trend focusing on the debugging of distributed programs has emerged. While this trend might lead one to suppose that the problem of debugging sequential, compiled programs had been essentially solved — thus allowing researchers to turn their efforts to a more difficult problem — such is not in fact the case.

Rather there seems to have been de facto agreement that, the point of diminishing returns having been reached, the subject was likely to permit further progress only grudgingly, if at all.

As a result, recent work in the area of debugging has focused almost exclusively on either graphical interfaces for debuggers, integrated debuggers for interpreted languages, or debugging of distributed programs. While the more general problem of debugging distributed programs is indeed both interesting and important, and while improved graphical interfaces can certainly ease the chore of debugging, the fact remains that many of the fundamental problems of sequential debugging for compiled languages have not yet even begun to be addressed adequately.

Unless and until a time- and cost-effective methodology emerges for provable correctness, the software community will spend a significant fraction of its time debugging sequential, compiled programs, and the sequential portions of distributed programs.

Driven not only by the importance of relatively bug-free software in numerous critical applications, but also by the competitive advantage accruing to both producers and users of high-quality software in the commercial arena, the search for better tools and practical heuristics to ferret out bugs will continue.

This thesis gives an overview of the state of the art in sequential debugging (focusing primarily on debuggers for compiled languages) and provides frameworks for the analysis of any particular sequential debugger on any architecture and operating system.

The initial goal of this work was not to build a better sequential mousetrap, but rather to investigate debugging through the construction and use of a prototypical debugger for distributed programs. However, once we had come up with the design criteria, we realized that most of the features on our wishlist would also be extremely useful for debugging sequential programs, yet they were lacking on all sequential, compilation-oriented debuggers with which we were then acquainted. Therefore this author worked jointly with Wilson Ho to develop a sequential debugger (called "Dalek"[1]) incorporating these features. Dalek's purpose was to act as a vehicle for exploring various novel concepts in debugging, allowing us to test both the feasibility of their implementation as well as their ultimate value to the end user. At present, Dalek can only be used to debug programs written in the C language, although some support for C++ is provided as well. While

---

[1] A semi-mechanical alien life form that was stymied by Doctor Who in its efforts to *exterminate* everything in sight. The authoritative "Unix Programmers' Guide to the Galaxy" states,

> All reported inter-species dialogues with Daleks have followed a Markovian process terminating inevitably in repeated, high-volume exhortations on the part of the Daleks to " *exterminate* | *exterminate exterminate** ". It is the only species known to date for which termination of a conversational process is synonymous with termination of the life processes of any non-Dalek interlocutors.

mixed-language debugging would be a desirable extension, we chose not to pursue that tack since its feasibility has already been demonstrated by others [John81].

Because debugging has historically developed in such an ad hoc manner, with various (often proprietary) tools for different languages arising on diverse operating systems, there is something less than complete consensus regarding a single, unified state of the art. Although certain celebrated debuggers (e.g., Magpie and the SNOBOL4 environment) have capabilities similar to some of Dalek's, those capabilities rely on the existence of an interpretive or incrementally-compiled runtime environment. Thus in what follows we very briefly review the capabilities offered by representative debuggers for (statically) compiled languages in order to motivate the need for a tool like Dalek, and to provide a background against which its features can be contrasted.

Three debuggers that run under the UNIX operating system are particularly well known: adb [Mara79] was historically one of the first debuggers to be shipped as part of the UNIX distribution. It is oriented only toward assembly language, although it does recognize symbolic names. Dbx [DBX83] is more modern, and is a true high-level debugger for the C language. Gdb [Stall89], the GNU project debugger, is a more recent arrival. Its source code is distributed by Richard Stallman and the Free Software Foundation.

Let's establish some terminology at the outset: The term 'user' refers to a person developing and debugging a program. The term 'debugger' refers to a program he[2] uses

---

[2] The use of the masculine form of the pronoun should not be construed by the reader to be indicative of chauvinist tendencies on the part of the author. Rather the above grammatical situation in fact demonstrates either a flaw in the English language definition, or a (repeatable) manifestation of a bug in this particular implementation of English. However, new results in higher-order type theory offer encouragement that a solution may at last be attainable. Rest assured that a bug report will be sent promptly — once the designated maintainers of English can be located.

to assist the debugging. The debugger controls the execution of the user's application program, which executes as a separate process under many operating systems. Under other operating systems, although the debugger and the application program *appear* to execute as two separate processes, they are in fact the same process. We will use the term 'inferior' in referring to either the separate application process, or to the combined debugger-debuggee process when it is executing code on behalf of the application program.

0.1. Typical Features of Current Sequential Debuggers

Existing sequential debuggers typically provide the ability to set breakpoints in the inferior program's code. When the inferior process encounters a breakpoint, it is suspended and control is returned to the debugger. Implementation of breakpoints, although architecture-dependent, is usually accomplished by the debugger overwriting an existing machine instruction with a separate "breakpoint instruction", as in 680x0 and VAX machines. Naturally the original instruction is saved somewhere, and is restored and executed when appropriate. Alternatively, in some instruction sets, e.g., the Burroughs B205, each opcode contains a reserved bit. If the inferior executes an instruction in which that bit is set, the desired effect is achieved via a trap to the operating system, which then transfers control to the debugger. Normally at this stage the debugger notifies the user by printing a message, and then awaits entry of interactive commands.

A surprisingly non-universal feature is the ability to associate a block of commands (written in the debugger's language) with a particular breakpoint. For example, such a block might contain commands to print the value of an inferior variable and then reestablish interactive control. When the debugger regains control after the inferior reaches a breakpoint that is associated with such a command block, the debugger automatically executes those commands before it returns to interactive mode or allows the inferior process to proceed past that breakpoint.

Many debuggers support so-called conditional breakpoints, in which a breakpoint can optionally be associated with a conditional expression. In such a case, when the inferior executes the breakpoint and transfers control back to the debugger, the debugger then evaluates the associated conditional expression. If the Boolean result is false, then the debugger quietly resumes execution of the inferior, rather than first notifying the user and executing any associated breakpoint commands[3].

Note that "conditional breakpoint" is somewhat of a misnomer, since the inferior will always be suspended when encountering such a breakpoint. Rather, what is conditional is the notification of the user that the inferior has reached that particular breakpoint, and the possible execution of an associated block of breakpoint commands (if that capability is also provided).

Another feature available in almost all debuggers is the ability to single-step the inferior process. Whenever the user orders a single-step, the inferior is allowed to resume execution, but only executes its next "step" before suspending again. Typically the user has a choice of the granularity of these steps (machine instruction level or source language statement level), and may also be able to specify whether such steps are to descend into

[3] Dbx on Suns, unlike the version on VAX-Ultrix, does allow breakpoint commands. However, it has **not** integrated them with conditional breakpoints. The two features are mutually exclusive!

any subsequently called procedures, or instead are to remain at the level of the currently executing procedure.

The debugger's command language invariably provides some method allowing the user to examine the inferior's data whenever the inferior has been suspended and control has returned to the debugger. With modern, "symbolic" debuggers, inferior variables may be accessed by name, as opposed to burdening the user with the task of discovering a variable's address[4].

In addition to these abilities for passively monitoring the flow of control and the state of the inferior's data, the debugger's command syntax will allow the user to change the values of variables (and sometimes code) and to alter the flow of control (via jumps, premature returns, etc.).

0.2. Less Common Features of Current Sequential Debuggers

The ability for the user to define new variables for use during a debugging session can be extremely beneficial. Such variables should (for reasons discussed in the next paragraph) be accessible only to the debugger, and might serve as auxiliary counters, or to store copies of inferior variables and data structures. But surprisingly, this feature is unavailable on many debuggers, including the standard UNIX C debuggers, dbx and adb, and the two pre-eminent DOS C debuggers, CodeView and Turbo Debugger. Gdb, Saber-C, and VAX DEBUG do support such debugging variables.

---

[4] However, the debugger frequently lacks a mapping from symbolic names to runtime addresses unless the user had the foresight to compile the inferior program with a special command-line option.

Even when this ability is provided, the choice of **where** such data is stored can be critical, since any disturbance of the inferior's address space can potentially alter its behavior, especially if the inferior contains bugs involving errant pointers. As if that were not bad enough, the practice of locating such supplementary data in the inferior's address space also means that our supposedly reliable debugging data will be susceptible to corruption by the inferior process.

Shades of Heisenberg [Heis58] — not only might one's "measurement" of an object disturb it, but that object might also subsequently disturb our records of that measurement!

From the tone of mild disapproval expressed above, the reader may correctly surmise that Dalek (like gdb) locates such data in its own address space. As such, it is secure from being affected by the inferior process. Likewise, since storage was not reserved within the inferior's address space for that data, its presence in the debugger's address space cannot possibly influence the behavior of the inferior, should it reference errant pointers or be sensitive to subtleties in the memory allocation pattern. We will henceforth follow the GNU parlance in referring to such user-definable entities as "convenience variables".

"Tracing" is another less common feature. Although this term is overloaded, it typically means that an informative message is printed (the inferior may be suspended as well) whenever the inferior's actions cause its state to satisfy certain user-specified conditions.

The most common (and least selective) form of tracing is to print every source line as it is executed. In a more discriminating tracing mode, the debugger might print the name of every procedure as it was called by the inferior, along with the values of its actual parameters. A similar message might be printed on exiting each procedure, including (for functions) the value being returned.

Popular changes of state that various debuggers can trace include alteration of a variable’s value, execution of a given line of source code, or the satisfaction of an arbitrary Boolean expression. Even by today’s standards, a quite sophisticated form of tracing had already appeared by 1960 [Floy60]. Tracing, in one form or another, is supported in many modern symbolic debuggers.

0.3. Unique Features of Dalek

Having experienced frustration with existing UNIX debugging tools for compiled languages, we hoped that our prototype, Dalek, could remedy that situation to some degree. To minimize unnecessary development effort, Dalek was based on an existing debugger, gdb, whose source code is publicly available. Our approach was to incorporate new features into gdb, while still retaining its basic command syntax. To the best of our knowledge, none of the features we claim as unique to Dalek was previously available under other UNIX debuggers for sequential, compiled programs.

It is interesting to note that a significant amount of Dalek’s development effort occurred *after* most of these features were implemented — our notion of what constituted a truly viable *integration* of these features evolved in tandem with our experience as users of Dalek.

0.3.1. Fully Programmable Debugger Language

We felt the most glaring deficiency of current UNIX compilation-oriented debuggers to be the lack of a fully programmable interface. By "fully programmable", we essentially mean that all capabilities available in C or Pascal should have appropriate analogs within the debugger. Dalek thus provides **if** and **while** statements, blocks with local variables, and user defined debugger functions — parameterized blocks of debugger commands that can return a value.

As a useful adjunct, we also implemented a rich collection of predefined debugger functions. For example, one such predefined function, when invoked with a parameter of N, returns (as a string) the name of the procedure N levels above the currently selected procedure in the inferior's runtime stack.

0.3.2. Broadcast Breakpoints

Dalek provides a feature we call "broadcast breakpoints" to assist the user who desires to set many breakpoints quickly, possibly without full knowledge regarding their target locations.[5] Broadcast breakpoints are available in two different modes. In the first mode, the user can insert breakpoints en masse at the entry to every procedure whose name matches a given regular expression. Such breakpoints can share a common block of breakpoint commands specified by the user at the time that set of breakpoints is broadcast.

---

[5] This is not to imply that the breakpoints will be scattered at random points throughout the inferior's address space, but rather that the user may possess only *declarative* but not *procedural* knowledge regarding their desired locations.

In the second mode, the user specifies not a pattern for procedure names to match, but rather an address range and an opcode mnemonic. Breakpoints may then be written at every machine instruction having the desired opcode within that specified address range. Once again, each such breakpoint in that particular broadcast set will initially share the same block of breakpoint commands in common with every other member of that set.

The reader should be careful to distinguish these senses of the term "broadcast" from that used in the distributed programming culture to denote the act of sending messages to several different processes simultaneously.

0.3.3. Characterizing Program Behavior via Events

In our literature search we had encountered the notion of *events*, but only in conjunction with the debugging of distributed programs [Bate82]. Yet we felt that they provided a potentially very powerful method for characterizing the behavior of sequential programs as well.

Events are an abstraction mechanism by means of which the user constructs partial models of the inferior program's behavior (either its intended or undesirable forms of conduct), somehow "hooks" them onto the inferior, and then monitors the program's actual behavior in terms of these specified forms. Dalek supports a highly-developed notion of events, even allowing the user to define hierarchical events — a group of lower-level events with certain relationships among their attributes can be recognized as constituting a higher-level event.

Traces may be viewed as one very degenerate form of events, although it seems impossible to extend the concept to hierarchical traces. Another limitation of traces is that the action initiated upon satisfaction of some trace condition is usually limited to the printing of some hardcoded (i.e., fixed format) message. Also, unless such printed messages can be redirected to a file, the information they contain will be lost as soon as the messages scroll off the user's screen.

In contrast, the actions initiated by Dalek upon recognition of an event (or even by a "near miss") are arbitrarily powerful, limited only by the expressiveness of the debugger language. In this respect, even the simplest events in Dalek are more akin to what might be accomplished via conditional breakpoints with associated debugger commands than they are to traces[6].

The minimal action taken upon recognizing an occurrence of some event typically consists of the event instantiating the attributes of that particular instance by permanently recording values of relevant inferior variables. In Dalek, such attributes can also include values of certain debugger convenience variables, perhaps to associate this particular instance of the event with the $N^{\underline{th}}$ call on the currently executing procedure.

Even if the event's actions are limited to quietly recording inferior data in its attributes, that in itself can be quite an advantage when compared with tracing. In event-based debugging the user need not be subjected to screens filled with tracing output as events

---

[6] In fact, given a set of integrable primitives including breakpoints with command blocks, convenience variables, **if** and **while** statements, user-definable debugger functions, and (especially) the ability to form arbitrary data structures in the debugger's address space constructed from such convenience variables, events could be implemented directly by the user.

are encountered, through which he must laboriously wade, sifting for useful clues[7]. Rather, he may selectively browse an event history list at any time, viewing the values bound as attributes of each successive occurrence of the selected events.

0.4. Summary of the Work’s Contributions

- Dalek provides a fully programmable debugger language, including **if** and **while** statements, and supports user-definable functions, written in the debugger language. It also offers a rich collection of predefined debugger functions to accomplish specific objectives that would otherwise be impossible for the user to achieve programmatically.
- Dalek allows the user to define his own events reflecting the abstractions upon which his understanding of the inferior program is based. The user specifies event recognition by means of familiar, imperative, programming language constructs, rather than via a special pattern-matching, regular expression language.
- Dalek’s unique, dataflow approach allows hierarchical events to be specified and recognized easily.
- As the inferior executes, Dalek records all instances of events in a database as they occur. In addition to a browsing mode, the user is allowed programmatic access to this event history at any time. This enables him to formulate and test new hypotheses without the necessity of restarting the execution of the inferior process.

---

[7] Satterthwaite notes ". . . unselective debugging aids usually produce much more output than insight." [Satt75]

- Under Dalek, breakpoints can be broadcast to various targets with a single command. This lowers the cost (in terms of the user's time and effort) of exploring a debugging hypothesis.

- Support for *meta-debugging* is integrated into Dalek's environment: the user can set meta-breakpoints in blocks of debugger language code, can meta-single-step through such blocks, and can request a meta-backtrace to display which blocks of debugging code are currently stacked, e.g., perhaps a top-level, interactive **while** loop was **step**ping the inferior, causing it to hit a particular breakpoint, whose code then invoked a user-defined function.

- Experience with Dalek has raised a number of interesting language design issues. As discussed in Chapter 5, several of our ideas regarding these issues have already been incorporated into the existing Dalek language. A fundamental tension will necessarily arise between two of the debugging language constructs in *any* fully programmable debugger. Fortunately, the new models of debugging we treat in Chapter 6 are sophisticated enough to lead to a solution that resolves the tension between the aforementioned debugging language constructs.

- Based on the perspective we have achieved as designers of debugger languages, we are able to offer certain conclusions regarding the precise nature and formal structure of any debugging computation, and of the role played by the debugger language. These topics are covered in Section 8.4.

0.5. Organization of the Thesis

The remainder of this thesis is organized as follows:

In Chapter 1 we review a variety of debuggers, all of which have been implemented and some of which are currently in widespread use. We are concerned not with the specific commands for each debugger, but rather with differences in their feature sets, their styles of interaction with the user, or their structure. In addition, a framework for analyzing the structure of any debugger is outlined; it shows why most existing architecture-operating system pairs force a tradeoff between two desirable characteristics — those of safety and efficiency. Certain enhancements to architectures and operating systems are suggested.

We then embark upon a three-chapter sequence that presents Dalek, our prototypical debugger, as it appears to the user. A variety of debugging and program exploration contexts are introduced to serve as a series of backgrounds for this presentation. Where relevant, we make minor digressions in order to highlight implementation issues, or to compare a particular feature with its nearest counterparts in other debuggers. During this tour, the reader may find it helpful occasionally to refer to the Appendix, which contains an index and quick reference guide to Dalek's syntax.

Chapter 2 begins the tour by focusing on just a few of Dalek's relatively simple features. It provides a gentle introduction to the syntax of Dalek's language, and includes the obligatory example of a real-life bug that was tracked down using Dalek.

Chapter 3 covers events thoroughly, commencing with a conceptual overview, and proceeding to several concrete examples that illustrate the power of event-based debugging. As befits its intermediate position in the three-chapter sequence, some of the Dalek code is nontrivial. Finally, an application is described that utilizes a newly specified user-defined function in order to programmatically access and analyze information recorded earlier in the event history.

Chapter 4 concludes the user's view of Dalek by concentrating on some of its more advanced capabilities. The first section considers why it may be necessary for Dalek to stack partially-interpreted command blocks. Next, to help the user manage the complexity that may develop, we present Dalek's features that support meta-debugging. To round out the tour, we cover certain realistic debugging situations in which some of Dalek's predefined functions come into play. Although some of the situations are necessarily rather involved, they demonstrate the power of the debugging methodology supported and encouraged by Dalek.

Chapter 5 first offers a brief tour through the implementation of gdb, the GNU debugger upon which Dalek was based. In the following sections, we then describe in some detail how each of Dalek's major new features was realized in the implementation. These sections do *not* constitute a low-level, line-by-line walkthrough of Dalek's source code, but rather a review of the design decisions and tradeoffs, and of the resulting structure of Dalek's source code. This chapter also examines several semantic issues that were elicited during the design and implementation phases.

Chapter 6 highlights some specific problems brought about by the user's ability (and natural tendency) to use different Dalek features in conjunction. Language designers have been presented with similar problems in the past, where the interaction of different constructs can cause unintended and undesirable results.

This chapter has a substantial theoretical content. It presents a formal model for any particular debugging activity, by means of which not only can its realtime efficiency be characterized, but the formal semantics of its event history can be analyzed as well. In contrast to most other analyses of formal semantics, that of Chapter 6 introduces no new formal syntactic notation to obscure the underlying semantics. Instead, the analysis is carried out (rather clearly and precisely we think) entirely in natural language.

A solution to the problems arising from "code interference" is synthesized. Its efficiency in most debugging situations is shown to achieve a lower bound on the best possible performance, given a pragmatic specification of the goals of that particular debugging activity. Simultaneously, we are able to show that the results of applying that construction to any particular debugging situation exhibit a formal property we call "semantic integrity", whose meaning effectively captures the essence of the intuitive sense of those words.

Chapter 7 is a wide-ranging discussion of issues primarily involving debugging methodologies viewed in the context of the software life-cycle, and how features like those supported by Dalek can be used to advantage throughout that cycle. Since by this point, the reader will be more familiar with Dalek as well as with the intricacies of debugging, this chapter also includes brief sections containing a more penetrating comparison of Dalek's semantics to those of "comparable" debuggers.

Finally, Chapter 8 summarizes the work up to that point and highlights avenues for future research. We then offer certain conclusions regarding the precise nature and formal structure of any debugging computation. Whereas the analysis of Chapter 6 concentrated on the *micro*structure of the debugging computation and how it could inadvertently be disturbed by specific instances of code interference, the analysis of Section 8.4 instead addresses the *macro*structure of the debugging computation and how that computation can unduly be prolonged by inadequacies in the power or expressiveness of the debugging language. That analysis enables us to give a rigorous definition of "fully programmable" and to show the significance of its role in the debugging computation.