CHAPTER 8

Summary, Future Research, and Conclusions

8.1. The Problem of Debugging, Viewed in Perspective

It is likely that debugging — if not fallout from the *lack* of adequate debugging — will continue to consume considerable resources (both human and machine) for the foreseeable future. Yet despite this fact, debugging is not widely regarded as a substantive research topic. We feel that some in the computer science community may have acquired a disparaging attitude towards debugging, perhaps partly in an ad hominem reaction against any individual so inept as to introduce bugs into a program in the first place. Viewed in this light, debugging is a problem unique to an "underclass" and unworthy of the attention of the community as a whole. Fortunately, those with even minimal experience in software engineering and the development of large projects necessarily take a more realistic view of the matter. Yet many of these same researchers mistakenly assume that — with the advent of modern high-level, symbolic debugging tools, possibly coupled with user-friendly graphical interfaces — the best possible tools for the job of debugging have already been developed. While such an outlook does avoid criminalizing the programmer, it nevertheless casually sentences him to a lifetime at hard labor, if he is forced to depend on current software tools.

More to the merits of the subject though, it must be admitted that debugging has traditionally been approached in an extremely ad hoc manner, with various tools for different languages arising on diverse, often proprietary operating systems. Thus, it is entirely contingent factors of history and economics that have tended, artificially, to partition the problem space and that have acted to limit the diffusion of knowledge between the domains imposed as a result. Quite often, knowledge that *has* been gained is transmitted not as a coherent body of techniques, but rather as folklore and ad hoc methods, often passed on by word of mouth. Even progress in debugging research that is successfully published in the open literature sometimes fails to transcend its original, narrow domain. Often, this is due to the fact that the results are expressed relative to a source language whose syntax is inaccessible to readers in other domains. Alternatively, the specific assumptions, tradeoffs, and design choices made by the researchers tend to be strongly dependent on a particular operating system and its underlying hardware, thus making the results inapplicable elsewhere.

A far more substantive objection to debugging as a subject of research is that it seems impossible *formally* to specify any exact problem statements. The most galling example of this is the difficulty in coming up with an exact definition of 'bug'. We all know that the existence of a bug somehow implies a problem. Should the bug be identified with the *symptoms* of that problem or with its *cause*? Yet the symptoms may vary with different input data, different compilers, or different machines. Worse yet, a symptom may appear only intermittently, even when the same object code is run on the same machine and input data. So it would seem more productive to identify a bug with its cause than with any particular symptoms it may or may not manifest. How closely is the cause of a bug associated with some particular location in the source code? In general, even if we ignore bugs introduced by the compiler, it would be unwise to identify a bug with one point in the source code, because we may apply its *fix* at a different location or at several places. Alternatively, we may claim a bug's cause is distributed over several locations in the source code, yet apply its fix at only one point, distinct from all the others. What if two bugs seem invariably to mask each other's symptoms? Are there then indeed two bugs, or none at all? And how often have we heard the famous quote, "That's not a bug — that's a feature!" (or a documentation flaw or a user misunderstanding).

Given such a fuzzy problem space and fragmented domain, it should come as no surprise that appraisals of the quality of any proposed debugging "solution" will tend to be similarly inexact and provincial in outlook. In the final analysis, the most valid criterion for evaluating the worth of a tool may well be an empirical one — the degree of acceptance attained by it (or by its conceptual offspring) among the community of practitioners.

8.2. Review of the Work's Contributions, So Far

The successful development of *structured methodologies* for debugging seems to be crucially dependent on the availability of powerful tools for two reasons: Obviously, if no sufficiently powerful tool exists to implement a proposed debugging method, then such a method can make no claims to being successful, even in principle, because our primary measure of success is a *practical* one. A more subtle, but perhaps even more important, dependency is that lacking a certain degree of familiarity with sufficiently powerful tools, one is less likely to be capable of imagining new techniques for debugging in the first place — or of imagining new applications for pre-existing techniques.

Our work with Dalek has shown that a pragmatic approach — in which users employ familiar, imperative programming language constructs to specify the recognition criteria for events of their own choosing — is entirely feasible. In contrast to the approach of Bates and Wileden, source code need not be modified (even automatically), and several of Dalek's fully programmable features allow the overhead of recording numerous instances of superfluous, predefined events to be avoided. In all fairness, it is precisely by modifying source (or object) code that Bates (and others) reduce the overhead *per* event occurrence; nevertheless, we feel that *any* change in an executable image is an inherently unsound debugging practice, and such meddling should be minimized unless realtime constraints are present. More to the point, it is the *user* who is uniquely qualified to define events with attributes that permit him selectively to monitor the state of an inferior program, and with *semantics* that allow him to interpret its dynamic behavior in terms of whatever abstractions he finds are most appropriate.

Dalek supports dataflow methods for recognizing high-level events. This approach, in conjunction with its fully programmable debugger language, allows arbitrarily complicated recognition criteria for primitive or high-level events to be easily and naturally expressed. This stands in sharp contrast to previous approaches [Ridd79]. For example, methods based on regular event expressions, although arguably more elegant [Salo66], unfortunately lack computational power. Slightly more powerful approaches, e.g., those that employ generalized path expressions [Brue83] — although limited to recognizing context-free patterns of events — nevertheless verge on becoming syntactically unwieldy. Related approaches that attempt to specify program behavior through disguised forms of path expressions ([Baia86]) strive for more power, but achieve only obfuscation.

Dalek maintains a database in which all events, along with the values of their attributes, are recorded as they occur. This hybridization of dataflow and database access to events has been conceptually unified by syntactically presenting the database to the user as an ultra high-level event whose constituents can be manipulated by predefined functions analogous to those used in the code of genuine high-level events. There is an extremely high degree of integration between events and Dalek's other features. Not only does this permit programmatic access to search the database of recorded events, but it also allows arbitrary actions to be initiated in response to the recognition of an event or even to a "near miss". Dalek's meta-debugging features can assist a user in fine-tuning the code in his events. Capabilities such as meta-breakpoints and meta-single-stepping provide a perspective on the nature of the debugging computation that may be unmatched in any other debugger. Then again, although features such as the meta-backtrace would be useful on any other, sufficiently powerful debugger, the user would be unable to invoke such features at meaningful points in the computation unless that other debugger also supported either meta-breakpoints or a **pause** command.

Dalek also offers a rich collection of predefined debugger functions to accomplish specific objectives that would otherwise be impossible for the user to achieve programmatically. Naturally, these predefined functions can be used in conjunction with events as well. As shown in Figure 2-1, one can employ broadcast breakpoints whose command block, although physically common to all members, can nevertheless be customized for each particular breakpoint by including calls to generic predefined functions such as **$func**(). Figures 2-5 and 2-7 illustrated how both broadcast and singleton sets of breakpoints can programmatically be "named" as they are established and subsequently be enabled or disabled using those names.

The two modes of broadcast breakpoints currently available in Dalek give the user an unprecedented ability to easily bind his primitive events to the most effective locations in the inferior. The third mode, using **$scan_line**() as proposed in Section 7.3.2, especially in conjunction with a parse tree, might surpass even the power afforded by many interpretive environments, in certain respects. Still, barring hardware-assisted debugging, there is no substitute for a virtual machine whose inner workings lie open to the user.

Nevertheless, free-floating events can offer a tolerable alternative to the unlimited number of "data watchpoints" available when debugging interpreted code on a supportive virtual machine. In debugging situations where the user is unwilling to bear the cost of events that are entirely free-floating, the judicious use of loosely-bound events, as illustrated in Figure 6-10, allows him to mitigate, to varying degrees, the performance slowdown imposed by events that are fully free-floating. The interrupt priority model we introduced in Section 6.6.4, and then simulated in Figure 6-8, offers hope that even novices can employ various constructs for loosely-bound events safely and effectively, rather than being forced to grapple with the semantics of general, re-entrant coroutines. By allowing the height of the global interrupt fence to be adjusted, the model also provides a more structured mechanism than that of breakpoint set "names" for *effectively* enabling or disabling classes of breakpoint sets.

Although the interrupt priority model may well be the best possible way to *manage* collisions between loosely-bound watch loops, the preferred alternative is to preclude such collisions in the first place by dispensing with explicitly coded loops for loosely-bound watches altogether. Fortunately, we were able to show that the TOPDOG synthesis of Figure 6-9 offers virtually the same realtime performance as might be achieved through the use of loosely-bound watch loops, but without the possibility of the semantic interference that makes that type of watch construct so hazardous to the integrity of the debugging computation. By transforming what was formerly control flow between loosely-bound watch loops into data flow between the pseudo-watches and TOPDOG, we were able to prevent exactly that remaining, problematic class of "tunneling" situations whose collisions, unbeknownst to the user, could destroy the semantic integrity of the debugging computation.

For any particular debugging computation, we were able to show that, barring any analysis of the inferior’s code, TOPDOG (or a minor variant, if selective interpretation were available) would take the minimum possible number of *strides* during the course of that particular debugging computation, *independent* of the number of watchpoint registers available on the underlying hardware, and *independent* of the degree of shared memory or number of processors supported by the resident operating system. We were able to identify these strides with an important practical measure of performance, namely, the context switches imposed by the debugging computation on the underlying operating system whenever the debugger and the inferior execute as separate processes.

Armed with these results and capabilities, we feel that users are more than adequately prepared to develop new, practical debugging techniques on their own, and after a period of consolidation, to refine them into coherent debugging methodologies considerably more powerful than many employed to date.

Section 7.3 covered, in some detail, the roles of behavioral specification and modelling in the context of the software life cycle. We will not belabor those points here, except to assert once again that truly useful and widely used programs tend to evolve considerably faster than their specifications. Our approach to events makes it feasible for a user incrementally to alter the semantic coverage of an inferior program's characteristic events, thereby supporting the use of "lite validation" techniques throughout every phase of the software life cycle. These methods offer hope that a specification may finally be able to keep pace with the evolution of the program whose behavior it purports to prescribe.

8.3. Future Research

We have barely scratched the surface in the area of behavioral modelling and event-based debugging.

8.3.1. Mappings, and Inductive Methods to Support Deductive Proofs

One topic to pursue is the use of events to monitor an axiomatic or algebraic specification of an inferior's semantics, and perhaps even to displace such forms of specification for certain purposes.

Algebraic specifications would seem to map rather directly into events. Higher-level events could be used to mimic the composition of algebraic functions. Thus it would appear that events are powerful enough to map *onto* the algebraic specifications, that is, to "semantically cover" them.

The relation between events and axiomatic specifications at the procedure level is quite similar. But given axiomatic specifications at the statement-level, the connection seems less straightforward. After all, if we have pre- and post-conditions for every statement, then are we not "debugging" a program that has already been formally verified? The problem is that certain semantically vital properties of the program may not have been proved, perhaps because the theorem prover (human or otherwise) was unable to intuit the proper loop invariant at some point. Dalek could assist a user in coming up with the invariants required to permit a formal proof.

Consider the probable atomic terms that might be combined in various arrangements to produce the clauses of logical statements. Such logical statements would be viewed as candidates, each vying for the status of pre- or post-condition assertions at some particular statement in an axiomatic specification of the inferior program.

Dalek could be loaded with what were believed to be the most likely atomic terms comprising an assertion at some particular statement in the inferior program. These atomic terms would be disguised as primitive events. Multiple runs of the inferior could then be performed on different sets of test data. Clearly any primitive event (atomic assertion) that survived this empirical gauntlet (i.e., that was *invariably* true or false) would be a prime candidate for inclusion as one conjunctive clause in the desired loop invariant. However, a primitive event might exhibit *contingent* behavior — during different passes through the loop or different runs of the program — and still wind up as either the antecedent or the consequent in several implication statements that involved the other, given atomic terms. Thus high-level events covering all possible logical combinations of the given atomic terms (primitive events) would need to be defined as well (it would be trivial to generate such definitions automatically).

While it seems that this approach might prove a very effective *filter* for candidate assertions, there remains the question of its *efficiency*. Clearly if a scattergun approach were employed, and primitive events were defined to represent not only every variable in the inferior, but every auxiliary variable employed in the axiomatic specification as well, the computation would be constrained by practical limitations of time, even if sufficient memory were available.

Alternatively, if a more selective process were employed to choose a more manageable number of initial atomic terms, then *if* the necessary atomic terms were among those actually chosen, this might seem to constitute an *effective procedure* for determining the desired loop invariant. But two other factors must be considered as well: First, we must also somehow select the sets of input test data with which we will exercise the device. Second, even if we were to employ only a very few primitive events, but require each high-level event to relate the most recent instance, e.g., of its low-level ‘constituent_term_1 with *every* past occurrence of its low-level ‘constituent_term_2, a combinatorial explosion would eventually saturate the dataflow graph.

8.3.2. Debugging of Distributed Programs

Extending our work to the debugging of distributed programs is an equally broad subject. In fact, that is where we originally began what has become the present work. Not only does this area present formidable practical problems, but the fact that for distributed systems, time imposes only a partial, rather than a total ordering [Lamp78], has profound implications for the semantics of event-based approaches in general. One consequence of the lack of a total order is that, in many cases, it may be impossible for a high-level event to determine which of its constituent events occurred first.

Another consequence is an existential problem: How are we to distinguish between an event that has occurred remotely, but has not yet been *reported* to us by another node, and an event that has simply not yet occurred at that remote node? Or, to phrase it in the terminology of distributed systems, even for a distributed computation that eventually halts, trying to monitor its fairness and safety properties by checking certain conditions for "eventually" and its inverse "never" — *even over any finite interval* — can require a good deal of patience, since such questions may not be decidable until *after* the computation *is known* to have terminated.

8.3.3. Meta-Debugging

Meta-debugging appears, in some ways, to be a more compact research topic, although it might seem to suffer from the same drawback as debugging itself — namely, that exact, formal results are very hard to come by. Nevertheless, it should be pursued.

The semantics of re-entrant dataflow graphs, mentioned in Section 3.3, are undeniably an interesting research topic, but would seem to be of dubious practical value, at least for purposes of debugging. Still, the topic might demonstrate some elegant structure, and might perhaps even find an application in some other area.

8.3.4. An Ad Hoc Wish List

We have emphasized that the most appropriate standards by which to judge progress in the area of debugging may be practical ones. While in the past this emphasis has often resulted in the development of ad hoc methods and tools, such need not always be the case. Besides, although pursuing ad hoc approaches may lead *directly* merely to the development of a random collection of unrelated tools, the use of such tools may provide an impetus that later allows them (or their offspring) to be unified under one conceptual umbrella.

For a moment, let us indulge in just such an ad hoc approach. Our "wish list" for Dalek includes several items that would be fairly easy to implement, but so far, we have not had the luxury of enough spare time to devote to this sort of "mere development". Enhancement requests include:

- Support for a new datatype — the heterogenous *list*. While it would certainly be convenient for users to be able to employ such structures directly, availability of a list data type would also effectively allow any instance of an event to have an unlimited number of attributes. For example, a single attribute could be used to record a snapshot of the current structure of an entire binary tree in the inferior. The existence of lists would also allow us to implement some helpful new, predefined functions, as defined in the following items.

- Two new predefined functions, **$parms**(N) and **$locals**(N). Calls to these functions would return respectively a *list* of the actual parameters, or of the values of local variables, in relative frame N of the inferior's runtime stack. Note that if all the user desires is for these values to be printed to his screen, those capabilities are already available via the **info parms** and **info locals** commands. However, there are currently no hooks permitting *programmatic* access to this sort of generic information. One consequence is that although we may employ, e.g., the **info parms** command generically in the code for a set of broadcast breakpoints to effect a *visual* printout, it is currently impossible to *record* such information, generically, among an event's attributes.

- A few more new predefined functions to allow programmatic file access. Presumably **$fscanf**() might be similar to its counterpart in C, although in this case, the "format control string" might dictate the expected types to be read into a list it will return as a result. **$fwrite**() might need no such "control string", since any possible arguments would already come tagged with their types. A consistent printable output format, perhaps using brackets, should be chosen for convenience variables that are of type list or "list of lists".

- Section 7.3.2 mentioned a possible new predefined function that we called **$bind_commands**(). The capabilities of such a function would permit the use of events to be integrated into every phase of the software development life cycle — from initial specification and prototyping, through development, testing, and debugging, and even (and perhaps especially) during the relatively "mature" phases of maintenance and co-evolution with a user community. The event definitions, *as well as their binding points*, would be integrated, directly, in the same files as the source code, thus serving as a permanent means of documentation. Since the event directives would appear to the C compiler merely as comments, they would *not* have any effect on the resulting object code or executable image.

- Section 6.5 mentioned a possible **$eval**() function. Given a string, **$eval**() would try to interpret that string, assuming it to be a syntactically valid debugger command.

8.3.5. Outline of one Method for Event-Based Testing

Clearly much work remains to determine how best to apply an event-based debugger in the realm of software testing. We mention one avenue, in particular, that is suggested by one of the items on our ad hoc wish list. While one particular approach does not constitute an event-based testing methodology, nevertheless we feel this method may be indicative of the techniques that may one day be found there.

Having some new syntax, even on paper, suggests some possible extensions to pre-existing debugging techniques. Recall that "lite validation", as advocated in Section 7.3.3, would refine the traditional use of a validation suite so that rather than merely detecting a difference in an inferior's expected *external* behavior long *after* it had terminated, events would be used to catch deviant *internal* behavior, long before this might have resulted in any external sign and hopefully very near to the point of infraction. But to accomplish this, the user would need considerable *knowledge* about the inferior's behavior, as opposed to mere information. The "raw" information will readily be available as event attributes, but the user is responsible for writing code to *interpret* the significance of that information, since it will be at runtime, as each event occurs, that the body of the event must ultimately decide whether or not the inferior is behaving properly.

By using the proposed new predefined functions to read and write files, — and by leaning heavily on the WEB-like approach [Knut84] that **$bind_commands**() would enable — we can implement an extremely complete form of comparative validation. To generate the initial, "trusted" standard for later comparison, the user need only instrument the inferior with events and then run it on the set of test cases, sequentially writing the contents of the event history database out to a logging file after each run. Thus the user need not (yet) concern himself with what is right and wrong about an inferior's behavior, merely with what actually *is*.

Next, as he instruments the new, modified version of the inferior with the same events (assuming this unfortunate version did not "inherit" a characteristic set of events from the preceding version), the user can append a few lines of code to the end of each event, to compare in turn each of its attributes to the next value returned by a call to **$fread** ($logging_file). With such a method, the user can avoid some of the burden of coding his events so that they "semantically cover" the inferior’s behavior. Rather, he is free to "syntactically deluge" the source code, assuming, of course, that there exists a relatively trustworthy copy of an older version of the inferior. He is free to record, as attributes, a multitude of data, and then to decide *later*, during a comparison run, whether any deviation from the "standard" of behavior is indicative of a bug or whether it seems to be appropriate for the newer version to operate in that manner.

8.4. Formal Conclusions on the Nature of Debugging

8.4.1. How to Make a Debugger Language ‘Complete’

Recall that one item on our wish list was a new predefined function, **$eval**(). Given an object of type string as its parameter, the function would attempt to interpret that string, assuming it to be a valid, debugger command. Fortunately, due to the nature of gdb’s original implementation, it would be trivially easy to implement such a capability.

One desirable change to Dalek that would *not* be easy to implement would be to augment the semantics of **$eval**() to cause all the output generated as a result of interpreting its parameter/command to be redirected to somewhere *other* than the user’s screen. To better communicate the idea, we introduce some new syntax: the same meta-characters that are available for purposes of redirection in the UNIX shell. For example, we might wish to append three values to a list by:

set $command = "print *x*; print *y*; print *z*"
call $eval ($command) >> $mylist

The redirection meta-characters are essentially just syntactic sugar, although the functionality they imply is definitely not. We could dispense with them altogether; for example, **$eval**() might communicate the output from its parameter/command not through the redirection meta-characters, but instead could return it, as a string, in the ordinary manner one expects of a function. That is, the preceding example might instead be coded as:

call $append (**$eval** ($command), $mylist)

This would actually be far easier to implement, since it would not require *any* change in Dalek's current syntax. Therefore, for practical reasons, we would probably *not* support redirection through an entirely new syntactic device, but would instead follow the path of least resistance. Nevertheless, in what follows, it will clarify our exposition considerably to employ these devices on paper, since their use permits us to *compose* functions without *nesting* them. Although the latter is an elegant practice, we feel it is less straightforward, visually, for any reader quickly to comprehend a deeply nested construction, in contrast to one that has been syntactically linearized.

So what is the significance of this augmented **$eval**()? After all, quite a few of the enhancement requests proposed other new, predefined functions as well. For example, we explicitly stated that implementation of **$parms**() would effectively *plug a gap* in Dalek's current feature set. Certainly it might be fun to play with a plain vanilla **$eval**();

the proposed redirection capability might even conceivably be useful in some bizarre, debugging situation. Why then, should an *augmented* **$eval**() seem to warrant any special treatment?

To answer that question, it will be necessary to digress for quite a while. Rest assured, though, that the existence of such "gaps" in a debugger's feature set is *exactly* the common thread that binds together the seemingly separate elements of the present discussion.

Earlier, in Section 1.2.1, we described Johnson's work with a programmable debugger he called RAIDE, and its language, Dispel. Johnson's work was important, and the significance of his form of programmable, event-based debugging has yet to be fully digested by the Computer Science community. Many ideas similar to those of Johnson appeared independently in Dalek, before we were aware of the details of his work. Although Johnson's approach to debugging was practical, he also had theoretical concerns about the power of the debugging tool he had developed. We quote from [John78]: (The italics are due to the present author.)

> There is no obvious way of dealing with the question "Can RAIDE do x?" without actually producing Dispel utterances to accomplish the desired goal. *In other words, how can it be "proved" in some rigorous way that RAIDE's primitive actions are sufficient to carry out all conceivable debugging requests?* Many of the traditional debugging requests have been shown doable by construction thru [sic] the examples . . . *Thus, the question becomes one of proving that the primitives are sufficiently general and extendable to implement all "computable" debugging requests.* Since no existing debugging system has been shown to fulfill this theorem, in some sense RAIDE is at least as well designed as other debugging systems.

In referring to "primitives", Johnson means those basic, building blocks of the language that can be arranged by the user to form a "debugging procedure". In Dalek's terminology, such primitives would include all its commands and the **if** and **while** statements, as well as any predefined debugging functions. Any combination of such primitives would, of course, be a user-defined function.

Having posed these questions, although in somewhat informal terms, Johnson seems to have concluded that it was impossible formally to prove that any particular collection of "debugging primitives" was indeed "sufficiently general and extendable".[1]

It may be instructive to consider some concrete examples at this stage. We would like to be able to say that if the **$func**() predefined function as well as the **backtrace** command were removed from Dalek, then Dalek's collection of primitives would certainly not be "sufficiently general" to answer all "computable" debugging requests. For example, the user might have, as a particular objective, that he wishes to know, "Who called *foo*()?", and it would seem that we have just made it impossible for Dalek to answer that question. Ordinarily, the user could easily have answered that question "manually" merely by setting a breakpoint in *foo*() and asking for a **backtrace**. But is there, perhaps, some way that some very different algorithm might be able to accomplish this same objective programmatically? Indeed there is! In fact, it could even be accomplished "programmatically" in gdb. Here is the method: Suppose the inferior had 1000 other procedures, each capable of calling any procedure via pointers to functions. The user could manually (and individually) set a breakpoint at the top of every procedure, and issue it a customized command block. For example, the command block at the top of procedure *proc_xyz*() would:

[1] Like Dalek, RAIDE also had gaps in its feature set of which its implementor was aware. In Johnson's case, for whatever reasons, his primitives could not be composed to effect **$func (N)**; rather, it was only possible, programmatically, to examine **$func**(0) or **$func**(1). We assume RAIDE had some capability *visually* to display a backtrace.

set $proc_xyz_stacked = ++$total_procs_stacked

The user would then have to manually view a disassembled listing of *proc_xyz*() and set breakpoints at every return instruction. Those breakpoints would contain a command block to:

set $proc_xyz_stacked = 0; **set** $total_procs_stacked−−

Finally, the commands at the breakpoint in *foo*() would say:

```
printf "My caller is %s\n", \
        ($proc_abc_stacked == $total_procs_stacked) ? "proc_abc" : \
        ($proc_abd_stacked == $total_procs_stacked) ? "proc_abd" : \
                etc., almost ad infinitum
```

That "programmatic" method is both so devious and so arduous that it is unlikely the average user would ever discover it, let alone use it if he did.

As another example, the reader might be tempted to say that gdb has a major gap in its feature set because it lacks an **if** - **then** - **else** statement. Clearly the user manually can, e.g., inspect the value of some variable and then decide whether to resume the inferior indefinitely or just for one step. Thus, his manual method constitutes an "effective procedure", so this must qualify as a computable debugging request. Gdb has no such programmatic capability. But wait! Even if, for some reason, gdb did not support the ternary operator we have just exploited, gdb does support the "conditional breakpoint", which allows an entire block of commands to be executed if a given condition is true when the inferior reaches its associated breakpoint. We need merely define two command blocks to hold the bodies of the **then** and **else** clauses, then write two adjacent "NOP" machine instructions in an unused area of the inferior’s data segment. One would then set a breakpoint at the first "NOP", make it conditional on the desired test condition, and associate the **then** clause with that breakpoint. An additional command to **jump** back to the inferior's normal code would be appended to that clause. The **else** clause would be associated with the unconditional breakpoint at the second "NOP", and it, too, would end with a **jump** to restore the inferior to its proper course.

We desire to formalize the notion of a computable debugging request, and also somehow to establish whether a particular collection of primitive features is indeed sufficiently general and extendable to answer all such debugging requests. But first, we will find it useful to introduce a somewhat more restricted concept, namely, what we term the *completeness* of a set of primitives.

- We will say that any particular collection of debugging primitives is *weakly complete* if, by combining them into a debugging procedure and then invoking that procedure, the user can *programmatically* accomplish any particular objective that he could also achieve *manually*, that is, by engaging in an iterative cycle wherein he would: interactively issue appropriate commands and then visually observe the debugging output that appeared on his screen (possibly recording some notes), before beginning the cycle anew.

Our choice of the term "completeness" to describe the above property is not accidental — our usage is analogous to the sense of "completeness" as applied to any other formal system. In order to extend traditional completeness methods to successfully include the realm of *interactive* formal systems, we shall need to explicitly establish the correspondence between the language(s) of our interactive system and the language of a more conventional axiomatic system.

In our case, the interactive formal system is composed of a debugger controlling a particular inferior. The debugger accepts its input language in two different modes — manual and programmatic.[2] In general, the output language of an interactive system might be considered to include everything that appears on the user's screen, with the possible exception of any input commands that are echoed. In order to be more precise regarding the output language of a debugging system, we might choose to define it in terms of events, because (barring memory limitations) events are capable of capturing every minute detail of an inferior's logical timeline. Thus, we might consider the output language of a debugging system to include a specific sentence if and only if there is some set of "objective" events with which the inferior can be instrumented so that, after executing for some period, that sentence is produced when a sequential printout of all events in the event history list is requested.

We may view each "phrase" (i.e., individual event occurrence) in an output sentence as corresponding to an assertion about the inferior's behavior. As mentioned in Section 6.1, the semantics of the proposition corresponding to a particular occurrence of some primitive event will typically be localized to the contents of a few specific memory cells at a single instant on the inferior's timeline. However, consider a phrase in a sentence of the output language that reflects an instance of some high-level event: The semantics of the proposition corresponding to an occurrence of a high-level event will typically *not* be localized at a single point on the inferior's timeline. Rather, in most cases the semantics of a high-level event will correspond to some property that characterizes a preceding *substring* of that very same sentence in the output language. Thus (by transitivity), the

[2] In general, an interactive system might employ a language for its manual mode very different from that of its programmatic mode.

semantics of a high-level event will correspond to some property characterizing an *interval* of the inferior's timeline.

For example, an occurrence of the high-level event ‘match (see Section 3.3.1) having attribute ‘match_addr may be viewed as an assertion about an immediately preceding substring and interval: The initial endpoint of that interval coincides with the most recent occurrence of primitive event ‘malloc whose ‘addr attribute is equal to ‘match_addr. The terminal endpoint of that interval coincides with the very next occurrence of ‘free having an identical ‘addr attribute. Naturally, the logical timestamp of the interval's terminal endpoint corresponds to that of these particular instances of ‘free and ‘match. During this interval, any invocations by the inferior of *malloc*() or *free*() respectively did not return a value equal to ‘addr, or were not passed a parameter equal to ‘addr.

Thus, a sentence in the output language represents a series of assertions regarding the behavior of the inferior. Although it is possible that some phrase in an output sentence may correspond only to a trivial assertion, each phrase — and therefore the sentence that is their conjunction — will be *true* if the semantic integrity of the debugging computation has been preserved.

Because our usage of the term completeness means the completeness of one input mode *relative* to another, we can avoid the traditional Gödel-numbering scheme by arguing as follows: Suppose that some collection of debugging primitives fails to be weakly complete. By definition, this means that there is some sentence in the output language characterizing the inferior's behavior that the user knows to be true (i.e., valid), since that sentence was produced by manual mode inputs. Yet it will be impossible to derive this same sentence using any programmatic mode inputs. Thus there will be a true sentence that is expressible in the output language, yet unprovable within the debugging system's programmatic mode — thereby making that system incomplete. We note, for future reference, the obvious synonymy of *manually* with "interactively". We also invite the reader to consider, but only in the background, what possible relations there might be between *programmatic* and an antiquated term: "batch mode".

Unfortunately, even in these very early stages, we already have a problem. Our definition of weak completeness would be very difficult to work with, because of its placement of the burden of proof. So far, if we wish to show a debugging language to be incomplete, we must exhibit some particular debugging objective (i.e., sentence in its output language) that cannot be achieved programmatically *using any algorithm*. Even if the debugging language had only a few, limited capabilities, there are still quite a few algorithms one would have to prove incapable of achieving that particular objective. How could one ever prove that all possible classes of algorithms had been exhausted? We have exhibited two such algorithms that, although highly "impractical", nevertheless successfully achieved the user's particular objective. Both examples demonstrate the enormous burden that our first definition places on anyone who hopes to show a *lack* of completeness. We must therefore retreat to a more easily defensible position, one where the burden of proof will fall instead on those arguing the other side of the completeness question.

Therefore, we introduce the notion of *strong completeness*. In contrast to weak completeness, *strong completeness* will require of a debugger's programmatic mode not only that it be able *somehow* to achieve the same objectives (i.e., produce the same output sentences) that can be achieved in manual mode, but also that it *employ the same algorithm* when doing so. To avoid controversy regarding whether or not two equivalent implementations actually employ the same algorithm, we can make an assumption that is eminently justifiable on practical grounds: The user will judge whether or not the algorithm represented by his manual-mode solution can be implemented in programmatic mode. This decision will be an easy one, because the languages employed in the two modes are identical. Clearly if a debugger's language is (strongly) complete, then it must be weakly complete as well. Thus in treating strong completeness, we shall henceforth dispense with the qualifier and speak only of *completeness* — meaning strong completeness. We emphasize that the completeness of an interactive system is fundamentally a comparison between two *modes* of input. In our case, since Dalek has no graphical, mouse-driven interface, both input modes employ the same language. One might prefer an incomplete but powerful language to one that was complete but ineffectual, e.g., the language over alphabet "*", where the semantic interpretation of "*" is to do nothing.

Having finally settled on a way to characterize debugging languages, it is certainly appropriate to ask, "Why that particular measure?" If our response were that completeness is essentially a meaningless property, whose sole purpose is that it can be easily proven or disproven to hold, then that would make the whole notion sound dangerously ad hoc. In fact, our response is that completeness is an excellent *practical* measure of a debugging language or, for that matter, of any computing language.[3] To begin with, although *weak* completeness might be considered a better measure of the ultimate *power* of a language, we are concerned also with the *expressiveness* of a language. Specifically,

[3] For example, any user would eventually become frustrated by the incompleteness of languages for application programs such as *vi* on UNIX or *MacDraw* on a Macintosh.

if a user has already (laboriously) figured out one (manual) way *procedurally* to express a solution to some particular problem, it is counterproductive to make him struggle to search for a *different* algorithm if he wishes to achieve that same goal programmatically. A second indicator of practicality is textual *conciseness*. As the number of steps actually executed during some algorithm increases, we would hope that the user could express the programmatic version of that algorithm using considerably fewer keystrokes than would be required to implement it in manual mode. Clearly, our example of a programmatic "solution" to the question of "who called *foo*()" in gdb requires the number of keystrokes typed by the user to grow linearly with the number of procedures in the inferior, rather than with the size of his particular debugging objective. This provides further evidence that strong completeness is a better practical measure than weak completeness. Finally, a third indicator of practicality is that, again, as the number of steps actually taken during an algorithm increases, the total realtime consumed both in expressing the programmatic version of the algorithm and then executing it should tend to be significantly less than the realtime consumed to implement that algorithm in manual mode. So if our definition of *strong completeness* is accused of being ad hoc, we can at least point to a number of very different criteria that it will be seen to meet in practice.

Our immediate task is somehow to get a formal handle on a rather vague notion, namely, what Johnson termed the "computable debugging requests". Fortunately, this seems doable, in contrast to the class of "conceivable debugging requests", with which he began the passage we have cited. After all, it is difficult *not* to conceive of the request: "Find that bug!"[4] Bearing in mind our inability to pin down the meaning of "bug" in Section

[4] For that matter, the manual for CDB, the HP-UX debugger, states that its language includes the "F" command to both "Find and fix bug". Surprisingly, the manual then proceeds to describe this as "a useless but

8.1, this would seem to indicate that, at least regarding debugging matters, the class of requests that can be *conceived* may be larger than the class of requests that can be *computed*. The surface of the argument is light at this point, but the thread that underlies it is quite serious: So far, we have implicitly equated every possible debugging objective with some "output sentence". But are we really justified in assuming that the event history is capable of representing the result of achieving "any particular objective" related to debugging? In fact, the distinction between *conceivable* and *computable* is vital, for in the definition of *completeness* we gave, we were quite selective in our choice of words.

> Our sense of *completeness* now hinges on establishing a precise definition for "any particular objective" that the user can achieve *manually*.

Theoretical speculation is all very well and good, but let us detour, for a moment, just barely across the line into the realm of what should be considered the remotely practical. Perhaps by analysing the capabilities of a specific debugger we can get a better idea of the kinds of particular objectives it might be used to achieve. After all, this discussion was motivated by the existence of gaps in the feature sets of two different debuggers, both of which were "extendable" by the user, and both of which would otherwise seem to be quite powerful, even universal.

Dalek began life with a pre-existing feature set, namely that of gdb. Perhaps as designers we can be excused for proceeding, in a somewhat ad hoc manner, to furnish it with new "primitives", especially in the form of predefined functions, in response to perceived deficiences in its feature set. Although we have not done so indiscriminately, nonethe-

---

humorous command". On the contrary, it would seem to be an extremely useful command, but then, perhaps it had not yet been fully implemented by the time the manual went to press. We might consider adding this command to our own wishlist!

less Dalek's feature set is far too complex to admit of easy analysis. We turn next to Johnson's RAIDE system, and its language Dispel, in hopes of a concrete example to support our analysis. After all, Johnson states:

> The kernel of RAIDE contains a small set of primitives sufficient to implement all the *traditional* debugging actions. Unlike previous debugging systems which have been designed rather haphazardly, RAIDE's design is based on the concept of *minimum sufficiency*.

(The italics are due to the present author.)

But again, such real world tools appear far too intimidating. Let us take Johnson's suggestion, though, in constructing a paper debugger of our own. We shall proceed methodically, so as not to burden our subsequent analysis with needless complexity. Let us strive for *minimum sufficiency* in designing our set of primitive features. Now tools such as debuggers traditionally are not designed for general computing purposes, but rather are intended for special purposes. Therefore, it might behoove us to include certain application-specific features in our set of primitive capabilities. We shall began with a very small set of these features, and they will be very primitive indeed. We will call our paper debugger UNDERDOG.

First and foremost, we must be able to gain knowledge about the inferior's current state. Thus we require access to the contents of its address space, which we shall provide in the form of a predefined function:

- **$peek** (addr) will return the current contents at that inferior address.

Now this is a considerable improvement over no access at all, but so far, this allows us knowledge only of the inferior's *current* state. Since we desire knowledge about a whole *sequence* of such states, we need "execute access" in addition to being allowed to read.

- Therefore, we shall provide a second primitive capability to single **$step**() the inferior. We will specify that **$step**() is to return, as its value, the contents of the inferior's program counter register after the inferior has taken that step.

It is important to note that in adding this second primitive, we most emphatically do *not* grant the ability, under *any* circumstances, to write in (or over) the inferior's address space. **$step**() will *not* achieve its objective by writing a breakpoint in the inferior's codespace. Instead, we will assume access to a "trap flag", hidden somewhere in the CPU that runs the inferior. We assume further that *if* the CPU running the inferior has any other registers that are accessible to the inferior, then those registers are somehow memory-mapped into its address space, allowing us access to them as well via our first primitive, **$peek**().

- For the record, we may *later* choose to provide a third, special-purpose primitive, namely the capability to **$poke**() a value into any location in the inferior's address space.

Let us pause, at this point, with a feature set that seems to offer all we need in the way of application-specific features, but that can make no pretense of supporting even the simplest general purpose computations. We wish to consider the most trivial, possible case of completeness, namely, with only **$peek**() and **$step**(), is the feature set complete? We readily admit that it is *not* sufficiently powerful to be of any practical use whatsoever, but that question is not relevant to our argument at this stage.

But although this case so far is clearly *trivial*, it is not yet *well-defined*, since in our original definition of completeness, we presupposed *some* kind of ability to combine debugging primitives into a debugging procedure, and then to invoke that procedure. Let us therefore provide such a capability, but once again, in order to simplify our analysis, we shall be careful to keep it as rudimentary as possible.

- We introduce, *temporarily*, into our feature set, a command to **source** a file containing only instances of **$peek**() and **$step**(). It is immaterial whether such files are permitted to contain other **source** commands.

To fulfill the remaining assumptions implicit in our definition of completeness, we should also state that to *mention* a primitive is to *invoke* it.

This, then, is *a* well-defined trivial case, and perhaps even the most trivial possible such case. We now have a concrete setup allowing us to test this notion of the *completeness* of a feature set. We allow the user interactively to issue **$peek**() and **$step**() commands, and thereby, in *manual* mode, to "accomplish any particular objective". We then invite him to place **$peek**()and **$step**() commands in a file, in whatever arrangement he thinks is most suitable to achieve his goal, and then we **source** that file into our debugger in order to test whether, **programmatically**, we can achieve that same "particular objective". Naturally, we assume that before the start of the second run, the inferior is somehow magically restored to its original state, so that both experiments have the same initial conditions. If we accept the further proviso that the inferior is fully deterministic, insulated from randomness and isolated from any interrupts in its external environment, then it would seem that completeness *is* possible, in one very trivial case: For any objective, we need only issue the exact same commands, in exactly the same order, as were issued by the user during his first, interactive run.

But it may seem that, in simplifying matters to such a degree, we have thrown the baby out along with the bathwater. On the surface, it would appear that somewhere along the way, we have abstracted out of the situation the very essence of debugging. After all, does anyone (other than an amnesiac) ever find the *same* bug, using exactly the *same* sequence of commands, for a second time? To verify its repeatability, possibly, but certainly not to *find* it! Besides, by thus programmatically copying the actions taken by the user manually, are we not somehow cheating?

The reader should hopefully feel disappointed at this stage of the argument, and perhaps he should even feel cheated as well! After all, he has arduously followed the establishment of a formal setup and he expects something in return for his efforts. The reader should be assured that this setup has already accomplished the purpose for which it was established: We *intended* to throw the baby out with the bathwater, for it was only in so doing that we could accomplish our goal. To quote from a popular song,

> "You don't know what you've got 'til it's gone".

Well, now that it's gone, we intend to discover *exactly* what it is that seems to have been abstracted out of our formal setup: Let us consider more closely the "final states" of the two experiments. It is clear that, having prohibited random influences, the inferior must, *by definition*, be in the same final state after both experiments. But one point is, how have we *gained* that knowledge? The flip side of that point is this:

> Let us ask the *user*, as the final arbiter, whether or not, in *either* experiment, he was able to "accomplish the particular objective" he had originally specified. Is he satisfied with the results of the debugging computation?

To do so, requires that we inspect the "final state" of the *user*, as well as his other, intermediate states during the course of the interactive experiment: We confirm that he has obediently been following the iterative cycle implicitly prescribed for him in "manual mode", and yet each time, after issuing his interactive command, *no output whatsoever has appeared on his screen*!

It is most emphatically *not* the case, that by cheating, by mimicking the user's actions "after the fact", we were able programmatically to achieve his goal. Rather, *neither experiment* has achieved *any* particular objective, at least none related to debugging. Although we have used a debugger and that use has propelled the inferior for some way along the natural course of its execution, we have been unable, in either mode, to gain any *empirical* knowledge of the inferior's behavior. It is precisely because the user has been unable to gain any knowledge of the inferior's behavior that programmatically mimicking his actions did not constitute cheating. Although such mimicry would be entirely legal if we wished only to show *weak* completeness, it could not, in general, be considered to implement the same algorithm. Indeed, UNDERDOG achieves *strong* completeness solely because the absence of output has restricted the class of algorithms that can be implemented in manual mode. Thus, what we know at the end of the debugging computation is exactly what we knew at the outset: e.g., that *if* you single-step an inferior eight times, *then* it will execute eight steps. After all, since nothing appears on our screen in response to repeated commands to **$peek**(), we have no direct, empirical knowledge of the inferior's state. In fact, it was only by *proof* that we were able, in the first place, to declare that the final states of the inferior were the same in both experiments.

> Why, then, have we been using a debugger — and not merely such a crude, primitive debugger, but apparently even a *buggy* debugger — to "accomplish a particular objective" that was amenable to a formal proof?

By examining this case a little more closely, we can discern why it is that UNDERDOG is complete, whereas neither RAIDE nor Dalek enjoy that property. Those real world debuggers are clearly far more powerful, yet they are incomplete. Could it be that the results of Gödel's famous First Incompleteness Theorem [Göde31] also apply to a comparison between two different modes of using the same formal system? That is, perhaps a lack of completeness is necessarily the price one pays once a certain degree of power is attained? Let us hope not!

The reader must forgive us if we seem to be beating a very dead horse. For ultimately what we desire in this argument is not just to prove that a useless paper debugger demonstrates an irrelevant property. Rather we are concerned with the nature of the debugging computation in the fullest sense, and with the respective roles of the human and the debugger during the course of that activity.

Had we been concerned *only* with completeness, we would not have gone to such excruciating pains to specify so precisely a trivial case that, we claim, demonstrates that property. Instead, we could have waved our hands: we could have chosen an *empty* feature set, and advanced that one, instead of UNDERDOG, as the trivial case. After all, if you can accomplish nothing manually and you can accomplish nothing programmatically, then does that not satisfy the criterion of completeness? But not only is it problematic whether or not that form of the trivial case is truly well-defined, in addition, it fails to shed any light either on the nature of completeness or on the essence of the debugging computation.

It should be apparent, by now, that we have a much better handle on at least two of the essential ingredients that got thrown out along with the bathwater:

> One essential element of debugging, as opposed to program verification, is that during (successful) debugging, not only is the state of the inferior altered during the course of the debugging computation, but the state of the *user* is altered as well,[5] in response to new, *empirical* data regarding the behavior of the inferior process.

> The other point regards the nature of completeness. We should investigate more closely *why* it is that UNDERDOG, itself, seems to have a bug. Since UNDERDOG is an idealized, paper debugger, it cannot be the case that the bug lies in its *implementation*. Rather, somehow that bug must have been inherent all along — in its *specification*!

We shall return to the first of our conclusions later. At this point, we wish to focus on this "bug" in the specification of UNDERDOG. Fortunately, the "User's Manual" for this beast is rather short. Let us debug the debugger. It has exactly three primitive features, two of which are specified as functions that are to return values when invoked. Believe it or not, this is the crux of what it means to be *complete*, in our sense of the term:

> We have failed adequately to specify *where* a primitive function should return its value.

The reader can be forgiven if he *assumed* that such return values were to be printed on the user's screen. After all, that is what one expects of virtually any system with interactive capability. But the specification for UNDERDOG says nothing about writing to the screen. For that matter, UNDERDOG might not even print a "prompt" when in use, nor is it necessarily true that it would even allow the characters typed in by the user to be echoed to the screen. In sum, UNDERDOG knows nothing whatsoever about the existence of the user's screen.

---

[5] Any alterations of the user's state should probably be limited to only those side-effects approved by the Coalition for a Drug Free America, if the semantic integrity of the user is to be preserved as well.

We could labor to augment UNDERDOG's feature set, giving it all manner of control transfer mechanisms, data structures, and procedural abstraction capabilities. We could, in fact, upgrade UNDERDOG into an application-specific Turing machine. Yet without some sort of primitive to **$print**() the values returned by the other primitives, the full power of such a machine is of no use to us whatsoever, either for general purpose computing or for its intended area of application, debugging.

We will leave UNDERDOG, for the moment, because by clarifying our understanding of the nature of completeness, it has accomplished its immediate expository purpose. Moreover, by illuminating at least one essential element of the human-machine interaction we call debugging, it has served us well.

Let us now venture further into the realm of the practical. We claim that completeness in any real-world debugging tool can also be ensured by supplying *only* a well-integrated collection of primitive debugging functions, convenience variables, an **if** statement, and the ability to combine these features into recursive debugging functions. But we must be very careful to state two additional specifications.

> We have learned, somewhat, from our experience with UNDERDOG, so we now require the existence of a primitive function that can **$print**() the value of its parameter to the screen. But hopefully, we have learned something else from UNDERDOG as well. Where previously we strove for minimum sufficiency in order to simplify the *analysis* of a debugger, although it may seem paradoxical, we must now strive for an absolute minimum of sufficiency, in a certain sense, if we are to *preserve* this property of completeness that we have carried so laboriously from that boundary between the theoretical and the practical.
>
> We must be quite specific in stating that it is *only* this single primitive function, **$print**(), which will be capable, *as a side-effect*, of writing to the user's screen. It must be made *impossible* for any other primitive feature to do so, except when

composed with **$print**().

We readily concede that the resulting syntax could be somewhat awkward. Nevertheless, by generating the interactive "command" set from the set of primitive functions, we can ensure the completeness of *any* well-integrated set of primitive debugging functions. Although the resulting command set might be cumbersome to use, it would not, in general, be trivial in its power.

Now how can this claim be proven? It should at least be apparent that, given any set of features satisfying such conditions, one could then combine the predefined functions into user-defined functions of his own choosing to extend their functionality to achieve that of the missing set of more conventional, high-level commands. What may not be so readily apparent, though, is that our restriction on writing to the screen is sufficient to guarantee completeness.

We invite the reader to consider a concrete example, namely Dalek. As matters currently stand, Dalek's set of primitive features includes the **info parms** command, but lacks the proposed **$parms**() predefined function. Mindful of our experience with UNDERDOG, we realize that this is significant: It means that there exists a command to "return" certain information to the user's screen, but that there is no way to route similar information to any programmatic construct. Thus the user, in *manual mode*, can visually inspect the information provided by **info parms**; he can write it down, and he can also cause the course of his manually driven debugging computation to "branch" based on what he sees. In contrast, because such information is inaccessible in *programmatic mode*, it can neither be recorded in convenience variables, nor be denoted within the conditional test of an **if** statement.

Therefore, for *any* system (e.g., *vi* or *MacDraw*), whenever programmatic mode is denied *equal access* to certain information that is available to the user visually in manual mode, then there exist manual algorithms exploiting that information that cannot be realized programmatically, thus making that system incomplete.

We have shown that we can achieve completeness with *any* integrated set of primitives, essentially by throwing away exactly those primitives that write to the user's screen. We would then introduce **$print**() into the feature set, and hope for the best. That is, although we can ensure *completeness* of the resulting feature set, we can offer no such guarantees that the resulting *power* of the machine will be anywhere near comparable to its original power when operated in manual mode. Thus this first method should be considered a *destructive* construction: We achieve equal power between manual and programmatic modes by reducing the manual capabilities to a lowest common denominator.

Clearly our first method of "completing" a feature set is not entirely satisfactory. Is there, perhaps, some way that the power of the two modes could be equalized by boosting the power of programmatic mode, rather than by attenuating the power of manual mode? Indeed there is. We need not prohibit any transfer of information between the machine and the user's screen; rather, we require one more predefined function that is, in some sense, the inverse of **$print**(): We require the ability, in essence, to **$read_screen**(). For it is only through such a capability that we can *programmatically* gain access to the same information that becomes available to the user during the course of the debugging computation. Of course, it would be extremely difficult to implement a *portable* version of such a predefined function.

Therefore, having digressed at length in order to paint the theoretical background, we are finally ready to connect the impressionist dots in that background to one of our earlier, ad hoc ideas: The second, and far more satisfying solution to our completeness dilemma lies in the idea we mentioned earlier, of supporting *output redirection* via a function like **$eval**(). We require only one other predefined function, **$sed**(), that will behave much like the UNIX stream editor of the same name.

We can now offer a practical method for achieving completeness that is entirely constructive. We shall remove nothing from an existing feature set. Rather, we shall *augment* that set with **$eval**() and **$sed**(), and especially with the ability to redirect the output resulting from the execution of any commands. Although we employ the metacharacters to more easily express that redirection, we remind the reader that these are merely a convenient syntactic device. The key is in the semantics of that redirective ability, not in any particular syntactic form by which that capability is expressed. Our use of **$eval**() is solely to remind the reader of that fact, and we emphasize that a string invocation capability is not necessary in order to complete a feature set.

For example, given Dalek's set of commands, its **if** statement, convenience variables, and the ability to create user-defined functions — along with the redirection capability — then even without any predefined functions except **$eval**() and **$sed**(), we can achieve completeness in the fullest possible sense of the word.

Now, armed with **$eval**() and **$sed**(), the user need no longer suffer due to the lack of foresight on the part of Dalek's developers in not providing a predefined **$parms**() function. Instead, being in possession of a complete set of primitives, he could write it himself, as shown in Figure 8-1.

By similarly executing commands, redirecting the resulting output that was formerly destined for the screen, and then filtering it, we are guaranteed programmatic access to *exactly* the same information, presented in the same format, that is available to the user operating in "manual" mode. That any debugger's feature set can be thus constructively completed, quod erat demonstrandum.

Thus, from some initial, ad hoc musings, we seem to have stumbled onto a unifying theoretical framework. Along those lines, notice that by adding **$sed**() as a primitive, we would no longer need to implement the *list* data type at all, because the user could instead manipulate the printable representation of any "list" directly as a string, e.g., using **$sed**() to select the N^{th} "element", i.e., character or word. As for the remaining

---

```
udf $my_parms ($relative_frame)
   # Generically return a list of the actual parameters in the specified frame.
   local  $pattern
   up  $relative_frame
   set  $pattern = "command_string_for_stream_editor"
   func-return  $eval ("info parms")  |  $sed($pattern)
   # $sed() will strip out verbiage regarding parameter names and types.
   # The  "|"  represents the UNIX "pipe" character.  If we wish to replace that
   # redirection character by function composition, the above could be written:
   # func-return  $sed ( $eval ("info parms"), $pattern )
end
```

Fig. 8-1 Example of programmatic access to visual information.

---

item on our original wishlist, should a new version of gdb someday appear with commands that allow writes to files, and writes from files to the screen, **$eval**() and **$sed**() would be fully capable of exploiting those new abilities in programmatic modes. In fact, since the **list** command inherited by Dalek from gdb already reads C language source files and writes their contents to the screen, the user could employ **$eval**() and **$sed**() to construct his own versions of **$scan_line**() and **$bind_commands**(), the new predefined functions proposed in Section 7.3.2.

8.4.2. Computability and ‘Fully Programmable’ Debugger Languages

We began our work on debugging two years ago in much the same way we began the journey of the last section — by reflecting on the contents of an ad hoc list of enhancement requests. We speculated about those "gaps" in the toolset, and as designers, we proceeded — albeit without formal, theoretical guidance — to plug those glaring deficiencies in the language. Yet two years after we began that task, Dalek’s feature set was still not *complete* in our formal sense. Fortunately, we have finally learned how to ensure completeness with the addition of **$eval**() and **$sed**(). But although completeness has proven to be a very useful concept, it appears that we still need something more. After all, even a rudimentary debugger like UNDERDOG was able to satisfy our formal definition of completeness.

Indeed, the completeness of an interactive system has — so far — seemed to be a very restricted concept. For our debuggers, it compared only the degree of expressiveness between two different modes of the same input language. What we would like now is some absolute standard with which to compare the power of an arbitrary debugger language. After all, there are many debuggers whose languages are so impoverished that whether or not they are complete is of little consequence. Perhaps it is time to approach "full programmability" with the same sort of rigorous analysis that paid off in our investigation of completeness. From the beginning (both temporally and textually) of this work, we have been slinging around the term, "full programmability" rather loosely, yet not once have we ever attempted to give a precise definition of that term. Earlier, we intimated that to achieve that (undefined) level of power would require the debugger language to have certain features like those of C or Pascal. As designers, we therefore added such capabilities to Dalek's language. We chose exactly the sorts of features from other languages that, when incorporated into Dalek, *should* have given it the capability of a universal Turing machine. And Dalek seemed to have that capability. Early on, for example, when playing around with Dalek, we wrote programs in the debugger language to implement the sieve of Eratosthenes and to decompose the Fibonacci series into its prime factors. What we did *not* know, however, was the exact relation between a universal Turing capability and debugging. We knew only that debugging was hard and that obviously we wanted to give the user as much power as possible. Yet there seemed to be a contradiction: How could a supposedly universal Turing machine still have such obvious gaps in its feature set? In retrospect, it is now clear why a lack of completeness hampered our universal Turing machine. After all, what good is programmatic computing power if it is denied access to the necessary data with which to operate?

At this stage of the argument, we set for ourselves a rather ambitious task: Not only shall we give a precise definition of the term *fully programmable*, but we believe the reader will be thoroughly satisfied that the definition we give is an intuitively plausible one. In the process, we shall simultaneously answer the challenge, posed earlier by Johnson:

> Thus, the question becomes one of proving that the primitives are sufficiently general and extendable to implement all "computable" debugging requests.

We sharpen our conceptual tools in preparation for the effort, but we discover that the very nature of the challenge requires, as an initial lemma, that we restate Church's Thesis in a rather unusual form:

> **Given** only the most rudimentary debugging capabilities, i.e., a debugger that can single-step the inferior (forward only; no "reverse gear" short of restarting the inferior), overwrite its code or data, and print its code or data on an (infinitely scrollable) screen, **then** the class of problems that the user can solve manually (given sufficient patience) is exactly that of the "computable debugging functions".

Now obviously, these computable debugging functions can include only functions that are themselves Turing computable, but if these debugging functions constitute a *proper* subset of the Turing computable functions, then how should that subset be characterized?

It will be helpful to view this lemma from a more concrete perspective: We have a Turing machine that we call the inferior. We choose to represent the state set of that Turing machine at a fairly high-level, namely, by means of (M + R)-ary tuples, where M is the number of virtual memory addresses available to the inferior and R is the number of registers. We shall consider the inferior to have multiple input and/or output tapes, each with an independently movable head. Such tapes will correspond to files the inferior may open, including standard input and output. Thus, we can give an instantaneous description of the inferior by specifying the values of the (M + R) quantities that represent its state and by specifying its cumulative I/O status.

To control that inferior, we introduce a primitive debugging machine, called UNDERDOG++, having the following capabilities:

- In lieu of permitting the user to take notes, we supply him with an infinitely scrollable screen.
- The debugger knows the entire initial contents of the inferior's tapes, as well as that Turing machine's starting state. Thus it is always capable of restoring the inferior Turing machine to its initial conditions, should the user desire it.
- The debugger allows the user, via its **peek** command, to read an arbitrary location in the inferior's address space. The specification of **peek** is identical to that in UNDERDOG, except that this command "returns" its value to the user's screen.
- The debugger allows the user, via its **step** command, to cause the inferior Turing machine to make a single move in its computation. The specification of **step** is also identical to that in UNDERDOG, except that again, the "return value" appears on the user's screen.
- Unlike its plain vanilla predecessor, UNDERDOG++ allows the user via its **poke** command, to write any value at an arbitrary location in the inferior's address space. Because this command may perturb the natural course of the inferior's computation, we discourage its use in most debugging situations. Nevertheless, it is available, should the user desire it.

As was the case with its predecessor, we emphasize that UNDERDOG++ achieves single-stepping by setting a "trap" flag in the inferior's CPU. Thus we need not be concerned about unintentional Heisenberg effects caused by the user setting a breakpoint that is subsequently read, rather than merely being executed by the inferior.

It may also be instructive to notice what capabilities are lacking in this setup. First, UNDERDOG++ per se has no programmatic capability whatsoever. Second, an extremely important point to keep in mind is that we do not know *how many* different internal states the inferior will enter during the course of its computation. For example, if the inferior uses a W-bit word, then it can have potentially $2^{W * (M + R)}$ states. It might be argued that any *particular* inferior has far fewer states, that is, it can have no more than $C * 2^{W * (D + R)}$ states where C is the number of addresses in the code segment and D is the number of addresses in the data segment. Thus we could give an instantaneous description by specifying only the I/O state, the program counter (and other) registers, and the contents of the data segment. But that formulation assumes that the inferior can neither read nor write its code segment, nor execute in its data segment. Suffice it to say that, in either case, our high-level description of the inferior as a Turing machine could still be enormously complex.

Because the state-entry problem is, in general, undecidable [Kain72], the best *any* debugger can do is to single-step the inferior and allow the user to inspect each state it enters during some finite interval. An obvious special case is that a debugger cannot determine whether an arbitrary inferior will ever enter the <HALT> state. One high-level consequence of the undecidability of the state-entry problem is that the only way in general to resolve, e.g., whether it is ever the case that (x == 3) for the inferior is, again, to single-step the inferior and watch that location. By attempting to answer such a question, we are actually attempting to decide whether or not the inferior ever enters an entire *class* of states.

It will soon become clear that what can be realized using such a setup are exactly the "Turing computable debugging functions". Thus we shall use UNDERDOG++ as a standard against which to measure all other debuggers. Given such a setup, we can now introduce a formal adversary argument. It may come as no surprise that the structure of this argument will bear a striking resemblance to that first, trivial, but extremely well-defined case of completeness we expressed using the original UNDERDOG.

We will treat the user, operating UNDERDOG++ *manually*, as an adversary. The rules of the game are that the user will pose a "debugging question" and then *he* will try to answer it, using only the manual capabilities we enumerated as the feature set of UNDERDOG++. Should he successfully answer the question, then *we* will be obliged to write a debugging program, using various constructs that may be supported by our own debugger, to answer the same question. While we would anticipate that such questions should typically relate to debugging or modelling the behavior of the inferior program with which we begin the game, it is not for us to judge. Note that should our adversary choose, he is permitted, albeit manually, to modify or extend a pre-existing program on a Turing machine. In conjunction with the infinite memory capacity of his scrollable screen and the ability to *restore* the inferior to its initial conditions whenever he desires, it is clear that he can realize any computable function *manually*. That is, even though the UNDERDOG++ machine per se is relatively feeble, it is still powerful enough to allow the user to access a *remote* universal Turing capability.

In particular, due to the user's infinite memory capacity and ability to single-step the inferior, it is possible for him to answer not only questions regarding *general* computable functions, but also certain application-specific questions regarding the natural (or slightly perturbed) course of the inferior's execution. Thus, we see that the class of "Turing computable debugging functions" may be exactly the same as the class of computable functions, if the adversary so wishes it. Typically, though, what will in practice distinguish the debugging functions from their less restricted brethren is that they will pertain to the behavior of a specific inferior or to minor variants of that inferior.

We might also add that, in order for either party to profit in a meaningful way from any such "debugging computation", it will help considerably to have access to certain information that is at a semantically higher level than that of bits and bytes. This makes the game more realistic and more relevant. Therefore we will allow access to certain higher-level information that will grant the user even more power to express his debugging questions as well as (possibly) to answer them. Thus, we shall provide initially, at the top of his screen, a printout of the inferior's symbol table (fortunately for him, the inferior has been compiled with the "-g" flag!), a listing of its source code marked with corresponding line numbers, and a description of the machine code for the target architecture. This, then, completes the setup of the problem.

Now suppose we measure "Dalek−−" against UNDERDOG++. That is, using Dalek, deprived of its **backtrace** command, its **$func**() predefined function, and also stripped of all its **info** commands, could we answer any question the adversary might be able to answer for himself by using UNDERDOG++?

Not surprisingly, the answer is a resounding, "No", despite the fact that Dalek−− has all the same primitive capabilities as UNDERDOG++, in addition to several powerful programmatic features that give it full Turing capability. Once again, the problem is one of *equal access* to information. The playing field established for our game with the adversary is simply not a level one. Using Dalek−−, although we can programmatically calculate the value of any Turing computable function, and although we have read, write, and execute access to the same inferior as does UNDERDOG++, we lack the knowledge of all those wonderful things that were printed initially on our adversary's screen!

The problem is that our adversary can pose, and then answer questions that are phrased in a language whose terms we simply do not understand. For example, the adversary can laboriously gain the information Dalek would normally provide in response to a **backtrace** command. Yet with Dalek––, we know nothing of how frames are implemented on the target architecture, nor of the mapping between source code lines and addresses in the inferior's code segment. For that matter, we do not even know the names of any of the inferior's functions! None of the "initial" information on the adversary's screen is stored in the inferior's address space. In UNIX, as in many other operating systems, the inferior's symbol table is never loaded as part of its executable image. And certainly, no self-respecting operating system would load the source code into that image. Instead, Dalek reads the symbol table into its *own* address space, and Dalek cannot **peek** at arbitrary locations inside itself. Rather, many of its primitive features can access that symbolic information in controlled ways. Similarly, other primitive features (such as the **list** command) can open the files containing the inferior's source code and display the contents in a controlled fashion.

Therefore, if any debugger is to be as powerful, even in manual mode, as UNDERDOG++, then it must allow us manual access to similar information as is initially available on the adversary's screen. Only then will we be able to carry on a discourse with our adversary in the language of the source code level, where we can speak meaningfully of coding *constructs*, procedural *abstractions*, and data *structures*.

In addition, if we wish to be able *programmatically* to answer all of our adversary's questions, then we need the output redirection and scanning capabilities of **$eval**() and **$sed**() as well. Without such capabilities, there will be no *programmatic* interface to that high-level language. Instead, although we would be able, *declaratively* to speak of high-level entities, we would lack *procedural* knowledge, e.g., the nitty-gritty details of exactly how frames are implemented on the target architecture. Thus the output redirection and scanning capabilities represented by **$eval**() and **$sed**() are essential if, having received our adversary's challenge, we wish to be able to convey that challenge to our user-defined functions without being forced manually to translate its high-level semantics into a machine-specific language of bits and bytes, *and* if, having received an "answer" from our programmatic constructs, we wish to be able to convey that response directly to our adversary without going through another tedious, manual translation process. In addition, if the events and user-defined functions we write are able to communicate *with each other* at a high level of discourse, then it becomes far easier for us to write them in the first place and to compose them in various ways in response to new questions from our adversary. To put it another way, if one is *only* allowed to use a high-level language while in interactive mode, then that will act as a powerful deterrent to discourage the use of programmatic or "batch" mode.

This, then, will be our definition of *full programmability*: We must be able to write a debugging program that realizes any Turing computable debugging function specified and implemented by the adversary, *especially* if the language constraining the relations between that function's inputs and its corresponding outputs includes terms referring to the higher-level semantic entities defined on the adversary's "initial" screen.

It may be appropriate at this point to reflect on one of the lessons learned during our encounter with the original UNDERDOG. The "result" of a Turing computable *debugging* function will not, in general, be a simple type; nor is it accurate to say that the result will be some particular "final" state of the inferior process. Rather, the result must also incorporate the *empirical* knowledge gained during the debugging session. Our adversary has a record of every bit of knowledge he gained on his infinitely scrollable screen. With Dalek's event history, we are able similarly to record exactly the desired information. Thus we can represent the result of a Turing computable debugging function as a cross product between the final state of the inferior and the state of the event history. This enables us formally to avoid any references to intensional states of the user.

8.4.3. The Debugging Computation is only *Incrementally* Specified

We began with the assumption that debugging was a hazy subject. We structured the process as an adversarial game and introduced sufficient terminology along the way to enable us to constrain some rather slippery concepts that might otherwise have squeezed between our fingers as we attempted to manipulate them. But most importantly, by operating with these more rigorous conceptual tools, we have managed to *localize* all that haziness that inhered in the subject of debugging: *It can now clearly be seen to stand where it has been standing all along — squarely in the court of our opponent.*

We imagine ourselves to be a fully programmable debugger: Whenever our opponent is *able* to pose a question — i.e., to *express* the riddle defining the next game in the series that *is* the debugging computation — we will be able *programmatically* to solve that riddle, without relying on any assistance from our "opponent", the user. Our opponent asks us specific questions, which (if Turing computable) we can always answer correctly. For our part, we too, make repeated requests of him; but the request we make is invariably the same: "Please state your next question".

The formal structure of the original game has not changed, only the intentions of its players. Debugging is not an adversarial situation, but rather a cooperative endeavor: We desire to keep the "rally" going as long as the user wishes to play. We are sympathetic towards our former opponent — there is some aspect to this game that he takes very seriously.

We will help in any way we can, which is considerable since we can realize any Turing computable function. In particular, we can realize that class of "Turing computable *debugging* functions", those which pertain to the behavior of *another* Turing machine, that is, to the behavior of the inferior process.

Unfortunately, the user is like a troubled baby — *something* is wrong with his world, but he is unable to tell us exactly what that is. He tells us that the goal of any interactive game played between people and computers should be to do good things, and that getting rid of bad things is a good thing. Further, he says the game we play is called "debugging", and that bugs are bad, and that the inferior has bugs.

Being a good, solid, Aristotelian machine, we are able immediately to deduce that the goal of "debugging" should be to rid the inferior of bugs. (So, we think to ourself, the syntactic similarity between "bug" and "debugging" is probably not a coincidence then, unlike so many other things in natural language. Fortunately, since "good" was only used as an intermediate term in the chain, we need not concern ourself with its exact meaning.) We are eager to assist the user in this game — he need only give us one more piece of information, and *if* the goal of this game we play is Turing computable, then we will reach it *without requiring any further dialogue* with the user. But the user is unable to provide us with this one, essential ingredient — he cannot give us a precise definition of "bug".

Nevertheless, we are patient with the user, since after all, he *is* capable of learning. He asks us a series of shorter, more limited questions, those which he *is* able to express precisely. If he is asking at least some of the right questions, then the answers we return will help him to formulate his final question — the one whose answer will end the game in a mutual win for both parties. In the meantime, we must be content to compute the result of this debugging function in stages, since its definition is only *incrementally specified* for us by the user.

We, as implementors of Dalek, began our quest in almost as hazy a state as that of the user we have just described. We knew, *vaguely*, that something in the UNIX toolset was inadequate, but we were unable to say just what that was. Fortunately, like the user, we were able to give that thing a name. He called the source of his troubles a "bug". As researchers, we named our problem "lack of full programmability". Although initially we did not know exactly what this "full programmability" was, we did at least know it when we saw it, and we saw that certain languages demonstrated that capability. So we incorporated similar features — by adding to Dalek's syntax mechanisms for control transfer such as **if** and **while** statements, and user defined functions. The ability to construct and to access arbitrary data structures was also provided to the user — not directly, but instead in a structured manner, through use of hierarchical events in a dataflow graph that always culminated in one "ultra-high-level" event, namely the event history database. After all, this was to be an *application-specific* Turing machine, was it not?

8.4.4. The Structure and Efficiency of the Debugging Computation

Exactly because "full programmability" *is* an application-specific concept, it should rightfully be defined within the overall context of the *debugging computation*, now that we understand that term more precisely:

> Full programmability means that *any* precise debugging question that the user can both pose and answer "manually" — i.e., using only the capabilities we enumerated in the command set of UNDERDOG++ — can also be coded in advance, thus enabling it to be answered *programmatically*, without requiring any intercession by the user during that particular game in the series. The debugging computation need *only* rendezvous with the top interactive command level in order to receive, from the user, the statement of its next incremental specification.
>
> Therefore for any computable debugging question, the ability to *express* that question implies the ability to write debugging code that will *answer* it programmatically.

Thus, just as the number of superfluous *strides* is a practical measure of how inefficiently a debugging computation is utilizing *machine* resources, the number of *unnecessary interactive stages* in the overall debugging computation provides a practical measure of how inefficiently the *human* resources are utilized in that same endeavor.

Perhaps it occurs to the reader that we may have come full circle: In the 1950's, before the existence of "operating systems", debugging was performed in batch mode. The "debugger" was a collection of subroutines, loaded from punchcards along with those comprising the "inferior"; together these constituted the sole program to inhabit the machine. Given the overhead, it was in the user's interest to code those debugging subroutines very carefully, since it would be a long time before he would have the opportunity to ask another question in the debugging series. With the advent of "timesharing", the lag between questions was reduced somewhat. Interactive debuggers reduced it even more, to the point where it became cheap enough that one could afford to ask even rather thoughtless questions, the sorts of questions for which the programmers of the 1950's would have been utterly unwilling to expend precious machine resources to solve. Rather, they would cheerfully have slaved away for hours, using only pencil, scratchpaper, and reams of printout.

Now with a permanent, event-based record of the inferior's behavior and full programmability, and especially knowledge of the precise *meaning* of that term we formerly used so loosely, we realize we have come not full circle, but rather 360 degrees along an upward *spiral* of debugging progress. A quote from Thoreau is highly appropriate here:

> Men have become the tools of their tools.

Or, as some anonymous sage put it more prosaically, "When the only tool you have is a hammer, you see every problem in terms of boards and nails". Exactly *because* it has become so cheap to ask questions, we tend to ask quite a lot of them during the course of the (now interactive) debugging computation. Unfortunately, with an increase in quantity has come a corresponding decrease in quality. Because there is no penalty in asking, our questions undoubtedly tend generally to be more stupid than those asked by programmers of the 1950's. We either fail to step back and reflect on the *meaning* of the responses we receive, or we are not permitted to frame our questions at a high enough level of discourse to profit from their answers. In either case, we are inundated by the flood of information comprising the responses to all our cheap questions and current debugging tools seem to offer no support for thoroughly *analyzing* the answers we have already received. This, then, is another consequence brought on by the style of interactive debugging supported by modern machines: By shortening the interval between questions, the cycle has itself been speeded up so radically that the human component of the process does not allow himself sufficient time to adequately perform his *own* function in the overall debugging computation.

Although the thought process during program implementation may be focused, line by line, at the level of the source code, the activity of algorithm and program *design* is certainly not. Instead, we employ pseudocode and procedural abstraction in order to manage the complexity in a top-down fashion. Debugging *ought* to be supported at the same level of abstraction. As an illustration, it was none other than (once again) John von Neumann who in 1947 [Gold47], in order better to reason about the behavior of computer programs, seems to have been the first to use global convenience variables to record, at "runtime", the number of times the flow of control reached certain points. Naturally he was not using an interactive debugger back then, but rather pencil and paper served as the "event history list", and he used his mind to single-step the inferior. If it was worth John von Neumann's time to engage in such activities *manually*, then perhaps the programmers of today might benefit from such a capability in machine-assisted form. Also, note that although John von Neumann (as well as any other sufficiently patient programmer) was competent to "intrude" on the task of the machine during the overall debugging computation, the machine is *not* competent to perform the task of expressing subgoals.

The need is not for machine resources to enable stupid questions to be answered cheaply, but rather is for software to support the phrasing of more high-level questions, as well as to aid in the analysis of the resulting responses. In this way, the machine can best assist the user in his task of setting intermediate goals.

We need a *methodology* by which to structure our interaction so that the debugging computation is efficient in terms of machine resources and productive in terms of human time. We need tools that, by making it cheap to do so, encourage us not only to *ask* questions, but also to *analyze* the responses — to *filter*, if need be, large quantities of such responses *programmatically* in order to free the user from that burden and allow him to concentrate on the task that *only* he can perform, namely, that of formulating the next subgoal in the series that is the debugging computation. It is unacceptable for those who claim to be professionals or computer scientists to debug by sprinkling "print" statements throughout their programs. The time of such people is too valuable to be squandered in such low-level pursuits. Instead, there should be available — within the UNIX community, at least — not only adequate high-level tools to assist them, but equally important, a methodology — the knowledge of how, safely and productively, to employ such tools.

8.5. Epilogue

We have argued in various sections throughout this work — and occasionally may have seemed to wave our hands rather strenuously — regarding the nature and fine structure of the *debugging computation*. We have advanced several claims regarding its various properties, in particular what we termed its *semantic integrity*. We showed that this could be influenced not only by direct, first-order, perturbation of the inferior's (more or less) natural course of execution, but especially by the so-called "meta-Heisenberg" effects — wherein our poorly-structured attempts at observation inadvertently alter the course of its *recorded history*, without necessarily disturbing the inferior's *actual behavior* in the slightest.

We investigated the class of "computable debugging functions" and by constructing an adversary argument, we were able to shed a little light on the ability of one Turing machine not only to monitor another, but especially to *interpret* debugging requests as well as the resulting observations in terms of a higher-level semantic framework that was meaningful to the user.

Use of that adversary argument enabled us to isolate the hidden, remaining cause of a lack of *full programmability* — and thereby to define that notion precisely.

We showed that without the ability *programmatically* to operate at a high enough level of discourse, then given any particular debugging computation, the number of *interactive stages* in that computation could be increased unnecessarily beyond what was required to compute that overall debugging function, despite the fact that it may have been only *incrementally specified* by the user, evolving dynamically, as it were, during the course of its specification.

That is, we showed that lacking full programmability, although the user may be able easily to *express* an *effective debugging procedure* to achieve a final goal or some particular subgoal, the debugger may be incapable of *computing* that debugging function in *batch* mode, but may instead require *interactive* assistance from the user. Thus, intercessions on the part of the user might need to occur, not only at those "evolutionary junctions" in the specification of the overall debugging computation (at which *only* the user is capable of setting the next subgoal), but might be required to achieve far lower-level objectives as well, even such mundane "goals" as allowing the inferior's execution to proceed until a particular variable was changed.

We noted the synonymy of *programmatic* with "batch mode" and realized that although what we refer to today as "the debugging computation" often takes only a matter of hours in realtime, that computation nonetheless has almost the same structure as that of its counterpart from the 1950's, which might typically have extended for several *weeks* in realtime. This temporal compression has altered the fine structure of the debugging computation slightly — it has shifted the balance between the roles of human and machine, so that often the human no longer has sufficient time to thoroughly ponder his "opponent's" last response before the opponent is back once again, asking for the next question. This, in itself, acts to interrupt the user's thought processes, but it is a necessary evil even though it may lead to the asking of stupid or superfluous questions.

What is *not* necessary though, is for the machine *prematurely* to return to interactive mode, well before it has completed the effective debugging procedure specified for it by the user, begging the user's *assistance* in order to compute those previous, precisely-expressed instructions. We claim to have proved that *any* debugger — running on any operating system and hardware platform — whose feature set achieves *full programmability* will make no such unnecessary impositions on the user's limited mental capabilities.

On the contrary — because full programmability implies not only the *power* of a Turing machine, but also (as initial conditions on that Turing machine's tape) some additional, high-level *knowledge* regarding the *semantic interpretation* to place on those bits and bytes that comprise the inferior — rather than the machine being *dependent* on the human for assistance in completing *its* assigned task, the machine can now hope to *assist* the human in completing *his* "assigned task", namely, the formulation of the next subgoal in the incremental specification of the debugging computation.

Thanks are due to Mark Scott Johnson for raising an interesting question in his Ph.D. thesis. By formalizing the notion of the debugging computation and by focusing on that class of computable functions that pertain to the (more or less "natural") behavior of another Turing machine (i.e., the inferior process), we were able satisfactorily not only to answer his *existential* question, but also to show how full programmability could be achieved in practice.

Although we claim to have sketched the outlines of several proofs, since, to paraphrase Imre Lakatos [Laka76], "Proof is in the eye of the beholder", it is up to the reader to judge the extent to which we have been successful in these endeavors.