Semantic Issues in the Design of Languages for Debugging

Richard H. Crawford, Ronald A. Olsson, W. Wilson Ho, and Christopher E. Wee
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562, U.S.A
debug@cs.ucdavis.edu

Abstract

Our objective in this paper is to survey some of the inherently difficult issues that will be faced by a designer of any imperative debugging language. Toward this end, we outline a powerful debugging language that we call GDL *(for General-purpose Debugging Language), justify the particular set of mechanisms that we have included in GDL, and address the issue of the minimality of this set. We focus especially on the semantic issues that arise when the language's mechanisms are combined—in short, the issue of being well-integrated. In our case, we see that GDL's mechanisms are well-integrated; however, some mechanisms are rather inefficient for many debugging applications. We, therefore, expand GDL's mechanisms, but in doing so, new semantic problems arise. We then show how these semantic problems can be avoided by following certain coding conventions.*

1 Introduction

This paper addresses some of the central semantic issues inherent in the design of imperative debugging languages. The key aspect that distinguishes a debugging language from conventional programming languages is that it must provide mechanisms that control execution of the program being debugged, allow exploration of that program, and, to be reasonably useful, allow these activities to be realized programmatically. By themselves, these requirements are nothing special. What makes designing debugging languages interesting and challenging is the *integration* of these requirements.

Languages for debugging have been largely neglected in the modern literature, which might lead one to conclude that no outstanding problems in the design of debugging languages remain to be solved. However, there has been negligible progress toward solving many fundamental problems of debugging since the advent of "source-level" debugging (i.e., debugging languages able to recognize entities present at the level of the source code language, e.g., statement labels and variable names). Moreover, the lack of attention to *languages* for debugging is primarily responsible for this lack of progress. Recent work in the debugging of sequential programs has focused almost exclusively on graphical interfaces to a set of underlying linguistic features that, in itself, has remained essentially unchanged for perhaps 20 years.

We have achieved significant progress in debugging by focusing initially on the debugger's language. Because a (sufficiently expressive) debugging language allows one to speak precisely and unambiguously about the low-level details of debugging, as well as to rigorously formulate higher level abstractions, this approach has provided an ideal vehicle for our research into debugging. That research has already resulted in a production-quality debugger, Dalek [11, 12], whose debugging language is quite respectable in its expressiveness and power. Once progress with a debugger's language has been made, the results should be amenable to "translation" – i.e., perhaps essentially identical semantics could be presented to the user packaged in a friendlier syntax, or provided via graphical modes.

Our objective in this paper is to survey some of the inherently difficult issues that will be faced by a designer of any *imperative* debugging language. We broaden our scope in the later sections to include certain *non-procedural* debugging language constructs. We shall generally speak as though our debugger is dealing with a *compiled* form of its target program — i.e., one that is executing directly on some underlying hardware platform. However, many of the semantic and language-design issues we raise are equally relevant to a target program that is being *interpreted* instead — i.e., running on a virtual machine composed of software, rather than of hardware and oper-

0-8186-2585-6/92 $03.00 © 1992 IEEE

ating system calls. Many of the tradeoffs we consider for implementation of the debugging language differ substantially between these two cases; our discussion highlights these differences.

The rest of this paper is organized as follows; additional details appear in [3]. Section 2 introduces relevant background material on debugging. Section 3 outlines a powerful debugging language that we call *GDL* (for General-purpose Debugging Language). Importantly, we justify the particular set of mechanisms that we have included in GDL. We also address the issue of the minimality of this set. Section 4 describes a key issue in the design of any language: semantic issues that arise when the language's mechanisms are combined—in short, the issue of being well-integrated. In our case, we see that GDL's mechanisms are well-integrated; however, some mechanisms are rather inefficient for many debugging applications. In Section 5, therefore, we expand GDL's mechanisms, but in doing so, new semantic problems arise. Section 6 shows how these semantic problems can be avoided by following certain coding conventions. Finally, Section 7 contains some reflections on our work and relates it to previous work.

2 Preliminaries: Architectural and System Support for Debugging

To anchor our discussion solidly in the real world, we make certain minimal assumptions about the systems architecture of the platform for which the debugging language is to be implemented. Such a framework is required regardless of whether the underlying platform consists of an operating system running a compiled target program directly on a physical CPU, or whether the target program is instead interpreted on another software virtual machine. We assume that the underlying platform offers no more than the four most common debugging primitives, namely the capability for a debugger to control and monitor its target process by:

- reading any memory location in the target's address space (including registers),
- writing any memory location in the target's address space (including registers),
- single-stepping the target process — i.e., causing it to execute a single (virtual-) machine instruction — without otherwise modifying its address space in any manner, and
- setting a code breakpoint anywhere in the target's address space.

For a *compiled* target, we shall assume that the debugger and its target program run as two different processes in separate address spaces. Although this *two-process* debugging paradigm corresponds to the situation in typical UNIX systems, it is not the only possible memory model for the debugging of compiled targets. For example, VMS runs VAX DEBUG [14], in conjunction with the target program, as a single process. The commingling of what otherwise would be separate address spaces is accomplished by linking the debugging routines directly into the target's executable file. However, this *single-process* debugging schema is more typical of the situation encountered when working with an *interpreted* target program, since in that case the interpreter, target routines, and debugging routines are usually all executed by the same process (e.g., [8]).

For compiled programs, we prefer [2, 3] the two-process model: it avoids possible perturbation of a target's behavior due to the presence of a debugger residing in the same address space and, conversely, it avoids potential corruption of the debugger, e.g., by errant pointers in the target. The runtime performance of the two-process model is, however, generally more costly than that of the single process model: a context switch is needed each time control transfers back and forth between the debugger and target processes. (Until [9], for a debugger even to read or write the target's address space necessitated a context switch.) We use the number of context switches as our primary metric of runtime efficiency. However, for targets that are *interpreted* rather than compiled, the debugger and the target will occupy the same address space. In this case, no context switch is necessary for control to transfer between the target's code and that of the debugging routines. Therefore, in a single-process debugging schema, the number of context switches is not a relevant criterion; instead, the efficiency of the primitives themselves is.

Finally, debugging requires support from compilers. For example, compilers must pass on symbol table information to debuggers. Besides the usual information that a compiler's symbol table may contain, the "debugging" symbol table also contains information that allows, for example, a source line number to be mapped to an address (or addresses) in the target process.

3 The Building Blocks of GDL

Below, we sketch the outline of GDL, a powerful language for debugging. The semantics of GDL were designed based on extensive debugging research conducted during the last several years. One outgrowth of that research effort was Dalek — a debugger distinguished from the previous "state of the art" in debugging primarily by its fully programmable language, and by the nature of its language support for user-definable events. Dalek's features also included provisions for representing hierarchical events, thus enabling users to monitor the behavior of their target programs in terms of the abstractions they found most relevant. The experience we and others have gained building and actually *using* the Dalek language has led us to a framework that captures the essential aspects of debugging, which is what GDL incorporates.

3.1 Special-Purpose Language Features

We begin our design of a debugging language by selecting primitives that are specific to the application domain of debugging. Perhaps not surprisingly, we choose four such primitives — *read*, *write*, *step*, and *break* — that correspond roughly to the four basic debugging capabilities supported by the hypothetical system architecture. At this stage, it is immaterial whether these special-purpose linguistic primitives have the syntactic form of statements, commands, or predefined functions.

An interesting issue is the question of the minimality of our primitives. Although *step* and *break* are not semantically equivalent, each can simulate the other.[3] (A *break* primitive requires an additional primitive to *resume* execution of the target.) The tradeoffs between *step* and *break* include efficiency concerns, whether code space must be modified (problematic for dynamic linking, shared code, and self-modifying code), and availability on existing systems.

3.2 General-Purpose Language Features

We choose to embed our special-purpose debugging features in a general-purpose language providing such familiar constructs as *if* statements, *while* loops, *block structure* allowing local and global *variables*, and library and user-definable procedures and *functions*. Let us suppose that all our special-purpose linguistic primitives are incorporated into this language as predefined functions, and let us allow the specific meanings of their incoming parameters and return values to remain unspecified for the moment.

3.3 Motivation for GDL's Feature Set

Some readers might question whether this language is too "rich" for our needs. One very basic premise of our philosophy is that the debugging language should be sufficiently powerful and expressive that it *encourages* a user to explore and analyze the state and behavior of the target program, rather than penalizing such attempts to glean further insights.

As an example, suppose at some point during debugging, the user desires to print out the contents of a linked list in the target program. If GDL lacked either a conditional statement or a looping construct, it would make the user reluctant even to undertake such an activity due to its tedious and manually repetitive nature. Similarly, the traversal of a binary tree structure in a target process becomes straightforward if GDL provides recursive functions and local variables.

On the other hand, some readers might claim that GDL is too low-level, since it lacks truly high-level constructs, such as a *print_linked_list()* predefined function. However, we feel that such routines should be provided in a library or written by the user, rather than cluttering the language itself with additional (not-at-all-)primitive constructs. The above example illustrates this point well: the code to traverse a linked list depends on the list's specific representation, i.e., whether it is circularly linked, has dummy head and tail nodes, etc.

Clearly, debugging variables, both local and global, are needed. It is convenient that they be dynamically typed.

3.4 Naming and Scope

A syntactic issue—*referents of names*— has a profound effect on the clarity of our subsequent presentation of semantic issues. Fundamentally, the issue is one of *scope*: Both static and dynamic scoping models are equally unsuitable for our purposes, because GDL must span two *conceptually disjoint namespaces*. Entities in GDL should not *mask* the existence of identically-named objects in the target process. Conversely, target objects should not mask GDL objects.

We, therefore, adopt the syntactic convention that names beginning with a '$' (or some other prefix not used in the source language of the target) refer to entities in the debugger's name space. This syntax allows us to make the *read* and *write* primitives implicit—their invocations driven by GDL's expression evaluation mechanism from positional context. This scheme is exactly that employed by the GNU project's gdb debugger [13]. We also follow gdb's convention of "reserving" a small number of variable names for denoting the *registers* of the target process, e.g., *$PC* for the Program Counter register, and *$SP* for the Stack Pointer register. For interpreted targets — where the underlying platform is a virtual machine — we assume that conceptual counterparts to *$PC* and *$SP* exist. See [3] for a more detailed discussion of the different syntactic possibilities and further justification of our choice.

4 Composing Higher Level Special-purpose Debugging Constructs

One design requirement of paramount importance is that all of GDL's disparate constructs should be well-integrated. We illustrate what well-integrated means in the context of debugging by two simple examples, which serve as a gentle introduction to more complex issues and compositions. Consider the following code:

```
while ( $SP >= $old_sp )
  $step();
```

It means that the target continues to execute until the current (i.e., current at the time this construct was initiated) function or procedure returns of its own accord. Consider now the following code:

```
while ( -1 != $step() )
  if ( $start_of_source_line($PC) )
    $print ( $source_text (
               $file_containing_address($PC),
               $line_containing_address($PC)
           ) );
```

It achieves the effect popularly known as "tracing" the target's execution. The loop terminates when the target process terminates. The functions *$source_text()*, *$start_of_source_line()*, etc. are provided as supporting library routines, which use symbol table information, rather than as GDL primitives.

The use of the while-step construction in the above examples for a *compiled* target incurs a significant runtime performance penalty given our two-process debugging paradigm, since its semantics logically necessitate a series of context switches as control is transferred back and forth between the debugger and the target.

The *$step()* primitive can be used to build higher level step functions. For example, the granularity of such step functions might be a "source_expression", a "source_statement", or even a "source_block". In addition, the behavior of stepping might be to step "into", "through", or "out_of". An actual implementation would require additional information from a compiler, but is readily achieved in an interpreter.

4.1 Importance of *while-step*

While-step constructs, as illustrated in the previous examples, provide a clean, simple mechanism for expressing and automating many useful debugging activities. Although (in the two-process model) their use entails substantial performance penalties, nevertheless it may be acceptable (and perhaps even desirable) in such cases for the user to leave the debugger unattended for a lengthy period to run in "batch" mode, its behavior governed only, but precisely, by the blocks of instructions comprising its GDL program.

The next subsections introduce two more applications for the while-step construct whose semantics are essential for solving many real-world debugging problems. Although alternative GDL constructs realize the same functionality, we express these applications in terms of while-step loops because this approach clarifies the absolutely fundamental nature of their semantics. In fact, minor variants of these applications comprise a *complete* set of lower-level debugging constructs. Later, we shall explore an alternative set of methods that minimizes the performance degradation incurred by the while-step construct for a compiled target.

4.1.1 A *while-step* Construct that Realizes the Semantics of a Code Breakpoint

The functionality of a code breakpoint can be simulated by means of the *$step()* primitive:

```
while ( $PC != $breakpoint_address )
  $step();
```

Because most architectures provide a breakpoint capability, our purpose in exhibiting this construction is primarily expository: It is important to realize that the *semantics* of the code breakpoint may be viewed as a *special case* of a more general semantic concept in debugging:

> Allow the target program to execute until a given, well-specified Boolean condition is satisfied.

In the case of a code breakpoint, the Boolean condition is merely that shown in the conditional test of the above *while* loop.

4.1.2 A *while-step* Construct that Realizes the Semantics of a Data Watchpoint

Figure 1 shows a while-step construct that allows one easily to implement an extremely useful debugging idiom: the so-called "data watchpoint". That term denotes some mechanism that causes execution of the target process to be suspended — whereupon control of the entire debugging computation is returned to the user — whenever the target process alters the contents of some particular, "watched" memory location.[1]

```
$x_ptr = (struct whatever_type *) &x;
$last_x = x;
while ( * $x_ptr == $last_x )
  $step ();
```

Figure 1: A GDL while-step construction to implement a global "data watchpoint".

Because the construction in Figure 1 is designed to watch a *global* variable ("x"), it is necessary to employ a debugger variable that points to it, rather than referring to "x" by name. In this manner we avoid potential scoping problems caused by *$step()*-ping the target into a block that declares a local variable also named "x".

By using GDL, one may employ various while-step loops to monitor every detail of a target process' behavior in terms of its Read, Write, and Execute modes of accessing memory (see Section 4.2.2 and [3]). In essence, these constructs satisfy the requirements for *low-level completeness* of a debugging language, since they allow a GDL program automatically to monitor the behavior of the target in sufficient detail to verify (i.e., prove) the reachability of any target state within a limited number of steps.

4.1.3 Combined Usage of *while-step* Loops

To conclude our "proof by demonstration" that certain GDL language features are indeed well-integrated, Figure 2 gives an example of how the various types of while-step constructs can be combined to monitor several independent aspects of a target program's behavior simultaneously.

```
$x_ptr = (struct whatever_type *) &x;
$last_x = x;
while ( -1 != $step () ) {
  if ( $PC == $breakpoint_address ) {
    $print ("Code breakpoint encountered at");
    $print (" 0x%x\n", $breakpoint_address);
    $commence_interactive_dialog();
  }
  if ( * $x_ptr != $last_x ) {
    $print ("Data watchpoint triggered for");
    $print (" variable 'x'.\n");
    $commence_interactive_dialog();
    /* "Reset" the watchpoint. */
    $last_x = * $x_ptr;
  }
}
```

Figure 2: Combined GDL *while-step* loops.

Figure 2 also contains calls to *$commence_interactive_dialog()*, another debugger library routine. This allows the user to engage in an interactive dialog with the debugger — perhaps inspecting the values of other variables, viewing the runtime stack of the target process, etc. — without having to exit all the way out to the top-level command interpreter. Instead, when the user chooses, he may return control to the (suspended) GDL block that issued the call to *$commence_interactive_dialog()*.

4.2 Efficient Alternatives to *while-step*

We now describe alternative constructs that employ the *$break()* primitive rather than *$step()* in order to achieve an improvement in relative runtime performance under the two-process debugging paradigm.

4.2.1 Binding GDL Code to a (Break)Point

A parameter of *$break()* specifies a block of GDL code; that block is executed as soon as control returns to the debugger after the target encounters the corresponding breakpoint. We say that the block of code is *bound* to the breakpoint address. For example, the code in Figure 3 counts the number of times control flows through a particular address in the target.

[1] Data watchpoints are not included as a GDL primitive because very few architectures provide them. The Intel 386 provides only four watchpoint registers; that is the most support commercially available. A few operating systems provide some support here, such as marking pages as Read-only; attempts to write to those pages are then trapped. Of course, most interpreters can watch an arbitrary number of variables at very little cost.

```
$break ( $breakpoint_address,
         "{ static $count = 0;
            $count = $count + 1;
            $resume(); }" );
```

Figure 3: Simple block of GDL "breakpoint code" *bound* to an address in the target process.

4.2.2 Implementation of Data Watchpoints via Broadcast Code Breakpoints

A few debuggers, notably VMS DEBUG and Dalek, allow the user to "broadcast" breakpoints en masse to all machine instructions in the target program having a particular opcode. Although GDL does not provide a separate debugging primitive to accomplish such broadcasting, by utilizing various debugger library routines, the user can compose one fairly easily.[3] The basic idea is to place a breakpoint at every "STORE-like" instruction; this can be accomplished using a while-loop that scans the code, disassembling instructions, and establishing breakpoints at each "STORE-like" instruction. Associated with each of those breakpoints, a block of code detects whether the watched variable has changed from last breakpoint. This approach, ignoring the initial setup cost, generally requires significantly fewer context switches than a while-step construct.

4.2.3 Semantics of Multiple Code Breakpoints at the Same Address

The ability to set multiple breakpoints at the same target address is desirable, e.g., to prevent a broadcast breakpoint from interfering with a conventional breakpoint. We therefore refine GDL's semantics for *$resume()* by specifying that it actually transfers control to the target *only* if it is invoked from the *last* breakpoint to be set at a particular target address; otherwise it transfers control to the code associated with the next breakpoint set at the same address. These enhanced semantics allow users to employ strategies involving broadcast breakpoints without concern for unexpected interactions. By contrast, most existent debuggers have notoriously ill-defined or undefined semantics in this regard.

5 Semantic Interference between the *$break()* and *$step()* Primitives

We have shown how every detail of a target program's behavior can be monitored by GDL constructs composed of the (implicit) *$read()* debugging primitive in conjunction with either the *$step()* exclusive-or the *$break()* debugging primitives. Thus, on architectures that provide a *$step()* debugging primitive but lack code breakpoints and data watchpoints, GDL can still implement these by using *$step()*. Similarly, on architectures that provide code *$break()*-points but lack a *$step()* debugging primitive and data watchpoints, GDL can still implement these by using *$break()*. But what if the underlying platform provides *both* of these debugging primitives? Might the user safely mix GDL constructs composed using both primitives, and if so, how? The answer is, "Yes, but *very* carefully"!

5.1 A *Free-Floating* Watch-Loop

The example in Figure 3 illustrated how a block of GDL code can be viewed as being *bound* to its corresponding target breakpoint address. A "program" in GDL might consist of several such "breakpoint blocks" *bound* to different addresses in the target, in addition to a series of GDL statements executed directly at the interactive command level. In contrast, a while-step block, rather than being bound to a breakpoint address in the target, is associated with an entire *range* of code addresses in the target. Another significant difference is that a top-level while-step block, although it has a single GDL entry point and a single GDL exit, causes the flow of control to depart GDL's address space entirely, before (presumably) having it re-enter the block. When a while-step construct — such as one designed to watch a global variable in the target — is issued from the debugger's interactive command level, we shall say that it is *free-floating* if it allows transfers of control to and from *any* address in the target's code space.

5.2 A *Loosely-Bound* Watch-Loop

In the two-process debugging paradigm, a context switch is required every time control is transferred between the debugger and the target process. This degrades runtime performance: a free-floating while-step loop taking strides the size of machine instructions incurs essentially the same performance penalty as setting breakpoints at *every* instruction in the target's code space. Although one should strive to eliminate any unnecessary context switches, this does not rule out while-step loops entirely in a two-process model. For example, if the user desires to watch a global variable in the target, s/he may be confident that its value is *not* altered within particular sections of the target's code whose execution accounts for a significant fraction of the target's total runtime. Conversely, the user may be confident that alterations to the watched global variable are issued *only* from within a few, narrow sections of the target's code space.

In such cases, for runtime efficiency in a two-process schema, the user may employ what we term a *loosely-bound* watch-loop. This is a simple variant of the while-step construct that is only active over restricted sections of the target's code, and whose lifetime thus spans considerably less than that of the target's full extent. In contrast to a free-floating watch-loop, a loosely-bound watch-loop is typically initiated within a block of GDL code that is bound to a breakpoint in the target. In addition a loosely-bound watch-loop normally terminates once the target's *$PC* advances beyond a certain point. The classic application for a loosely-bound watch-loop is to monitor changes to a *local* variable. One sets a breakpoint at the entry to the relevant procedure in the target, and then associates with it GDL code similar to that shown in Figure 4.

```
$break ( $entry_to_procedure("foo"),
         "{ static $current_sp = $SP
            static $last_x = x;
            while ( $SP >= $current_sp ) {
              $step();
              if (   $SP == $current_sp
                 &&  x != $last_x ) {
                $print ("local var changed");
                $print (" from %d to %d\n",
                          $last_x, x);
                $last_x = x;      } } }"  );
```

Figure 4: A *loosely-bound* GDL while-step loop to watch a *local* variable named "x".

5.3 Faulty Monitoring due to Collisions between Blocks of Code

Having introduced sufficient nomenclature, we can now characterize any GDL block as being bound, loosely-bound, or free-floating with respect to the target's code space. We now address the original topic of this section, namely, whether on architectures that provide both the *$step()* and *$break()* debugging primitives, it is possible to mix bound blocks with loosely-bound or free-floating blocks. Unfortunately, constructs based on one of these debugging primitives are *not* yet well-integrated with those based on the other primitive.

When used in isolation, each of our GDL constructs does indeed behave as advertised. But combining them may produce extremely misleading results. Consider what may be the simplest combination of GDL constructs into a GDL "program". Suppose we bind the block of breakpoint code shown in Figure 3 to a particular target address, and then employ the free-floating while-step loop of Figure 1 to monitor changes to a global variable. Assuming the flow of control in the target eventually reaches the breakpoint address, the watch-loop will *$step()* on the user's breakpoint — with results that are not (yet) covered by the language definition. If the while-step loop were to ignore the presence of the breakpoint, then (at the very least) the GDL block of Figure 3 would not be executed, resulting in an *incorrect* flow count past that point. This is obviously unacceptable.

Because the results of such a "collision" are clearly of vital importance for the integrity of the debugging computation, the GDL language definition *must* cover such situations, rather than leaving the semantics up to each particular GDL implementation to deal with in its own way. But what *are* the "right" semantics in this case? Our guiding light will be the Principle of Least Surprise. In the (quite typical) situation we have just described, it is clearly the user's *intention* to simultaneously monitor multiple unrelated aspects of the target program's behavior. We accommodate such desires by specifying that the GDL language definition requires that, whenever one GDL block causes a transfer of control to the target process — e.g., via the *$step()* primitive — execution in that GDL block does not terminate, but rather is suspended. Should the flow of control "immediately" return as anticipated — i.e., from the target directly to that same GDL block — then execution of that suspended GDL block will be resumed at the statement following the *$step()*. On the other hand, if, upon returning from the target process, the flow of control re-emerges in a *different* GDL block (typically because the target has encountered a breakpoint, or received a signal "trapped" by another GDL block), then the original GDL block is to remain in its suspended state, pending its eventual resumption.

But our simple GDL example "program" will still not behave as intended with these semantics. The key point is that, after the first breakpoint was encountered, the watch-loop has remained suspended, "asleep on the job", and thus has been oblivious to any changes in the target's state. More specifically, the while-step loop is suspended once it executes *$step()*, transferring control to the target process. The target encounters the breakpoint, suspends, and control then re-emerges in the block of GDL code associated with that breakpoint. The statements shown in Figure 3 are executed sequentially, with the last statement in that block invoking *$resume()*, thus initiating a transfer of control back to the target process. The user's intentions will not be correctly implemented, because after the first breakpoint is encountered, the while-step loop will remain suspended until the target actually terminates! Only after the target's termination (according to the GDL language definition) will the series of suspended GDL block activation records begin to "unwind".

However, our attempt to specify GDL's semantics has not been a failure. On the contrary, GDL's semantics are now sufficiently robust to handle even the most complex debugging activities. What our simple example demonstrates is rather a misunderstanding on the part of the user regarding certain details of a properly structured GDL program. Yet the nature of debugging is such that it is unrealistic to require the user to know, in advance, exactly how many and which specific aspects of the target's behavior s/he desires to monitor during runtime. Therefore, a user must be able *incrementally* to write a GDL "program" that is composed of *semantically independent* blocks — i.e., GDL blocks whose execution by the debugger will not *interfere* with the correct functioning of other GDL blocks. Thus a block of GDL code — such as a free-floating while-step loop — should function as intended regardless of the actions of subsequently written blocks of GDL code (e.g., a GDL block bound to a breakpoint).

6 Structured Programming in GDL

6.1 Conceptual GDL Program Structure

An invocation of the GDL *$break()* primitive does not result in an immediate transfer of control to the target; rather, it effectively implants a Remote Procedure Call to the debugger in the target's code. To the unwary user, the effect may be akin to "booby-trapping" the target program: If one sprinkles breakpoints throughout the target's code to monitor various activities, yet fails to follow a *disciplined* coding style when writing their associated GDL code, it will be almost impossible to predict the resulting trajectory taken by the single thread of control on its journey through the combined GDL-target program. Undisciplined use of breakpoints is harmful for exactly the same reasons that "GOTO" was considered harmful — because it leads to spaghetti code. In the absence of a disciplined GDL coding style, the appropriate paradigm for debugging is no longer a simple two-process master-slave model, but rather a model involving an arbitrary number of independent *co-routines* — since in effect, a block of GDL code may behave as an independent co-routine. The user can indeed be confident that, e.g., by executing a *$step()* primitive, control will transfer immediately to the target program. But it would be a mistake for the user to assume that the next transfer of control from the target to the debugger will be a "return" to that same block. Yet surely it should not be necessary to write a re-entrant GDL program in order to debug a sequential program that is executed by a single thread of control!

6.2 Ensuring Cooperation between Free-floating Watch-loops and Breakpoint Blocks

Interference between while-step loops and GDL code associated with breakpoints can be prevented by following a simple coding convention. Upon entry to a while-step loop, the user increments a global debugger variable (it is initially zero). When writing GDL breakpoint code, the user must then *guard* each call to *$resume()* by checking the value of that global variable, e.g.,

```
if ( $in_while_step_loop == 0 )  $resume();
```

If *$in_while_step_loop* is non-zero, execution will "fall off the tail" of that block of GDL breakpoint code, thus resuming execution in the suspended while-step loop.[2]

Naturally, the user must remember to include code at the loop exit to decrement *$in_while_step_loop*.

6.3 Preventing Interference between Loosely-bound Watch-loops

If the user employs several *strictly nested* loosely-bound watch-loops — e.g., if target procedure "foo"

[2]This semantics is quite consistent with the Principle of Least Surprise. If, when execution fell off the tail of some GDL breakpoint code, there were no underlying suspended while-step loop, then control would "land" at the top interactive level, engaging the user in dialog just as one would expect.

calls procedure "hoo", and two watch-loops monitor variables local to each procedure — then interference cannot occur. However, if there is even the slightest possibility that, at some point during the target's execution, both watch-loops *should* be active simultaneously, then either the user should recast all his constructs in terms of breakpoints and avoid *$step()* entirely, or else use an entirely different construction, as shown in Figure 5. This construct allows unrestricted use of code breakpoints, and allows the user to employ whatever quantity of "hardware" data watchpoints are provided by the underlying platform. Once those hardware resources are exhausted, the user may employ loosely-bound watch-loops, albeit only in a tightly-controlled fashion. In Figure 5, new watch-loops are not coded directly, but instead are "registered" with a single, top-level while-step loop, which is responsible for "waking" each watcher at the proper intervals during the target's execution.

One crucial difference is that watch-loops are no longer sovereign constructs that independently instruct the target to *$step()*, but rather have a *non-procedural* aspect that expresses what to do, but not how to do it. Thus, the single top-level loop is able to coordinate their requests in a manner that precludes inadvertent interference. Figure 5 also incorporates a construct to optimize runtime performance: It invokes *$step()* only within intervals of the target's execution which are of interest to registered watchers. During other periods, it invokes *$resume()*, allowing the target to run free at full speed until it encounters a breakpoint.

7 Discussion

One of the inspirations for our work is [6]. However, that work did not address the fundamental issues that we do in this paper. Further details on our work appear in [2].

Common existing debuggers (e.g., dbx [4], gdb, sdb [7], and VMS DEBUG) for compiled programs are not as attractive as GDL since their debugging languages are not as expressive; many of these debuggers also suffer from ill-defined or undefined semantics for combinations of their features. Some interpreters also provide substantial debugging aid (e.g., for Saber-C [8] and for SNOBOL [5]). These are successful, to a degree, in providing powerful debugging mechanisms, but have not explored the fundamental issues addressed in this paper.

Higher level abstractions, such as events [1, 10, 11, 12], are also useful for debugging. They are not just convenient, but essential for debugging large, realistically-complex target programs. Such higher level abstractions require a well-defined language on which they can be built. They are, however, *potential* sources for additional code interference, akin to the interference seen earlier between watch-loops and breakpoints. However, if the lower level mechanisms are well-integrated, which we described earlier how to do, then the higher levels can be well-integrated among themselves as well as with the lower level ones.

Our main concern in this paper is with the *semantics* of debugging, so any reasonable syntactic vehicle suffices if its semantics are clear and precise. We have chosen a C-style syntax, primarily for expository purposes. We choose *not* to address the valid question of whether the debugging language should appear syntactically similar to the target language, or whether it should be clearly distinguishable from the source language of the target program. That question is inextricably linked with conditions of use — e.g., whether the user community is comprised of expert or novice programmers, whether mixed-language debugging is to be supported, etc. In addition, for an interpreted target (where the underlying architectural platform is a virtual machine) there may be no choice but to use the same language for the debugging routines as was used to write the target program.

One interesting area of future research is to study the formal semantics of debugging languages such as GDL. Work on the semantics of coroutines or concurrent programming languages should be applicable.

Acknowledgements

We thank the anonymous referees for their useful comments on earlier drafts of this paper.

References

[1] B. Bruegge and P. Hibbard, "Generalized path expressions: A high level debugging mechanism," *Journal of Systems and Software*, Vol. 3, pp. 265-276, 1983.

[2] R.H. Crawford, *Topics in Behavioral Modelling and Event-Based Debugging*, M.S. Thesis, Div. of Computer Science, University of California, Davis, CSE-90-49, December 1990.

[3] R.H. Crawford, R.A. Olsson, W.W. Ho, and C.E. Wee, *Semantic Issues in the Design of Languages for Debugging*, Div. of Computer Science, University of California, Davis, CSE-92-7, February 1992.

```
while( $running ) {
  if ( $watchpoints_active  <=  $watchpoint_resources )
    $resume();    /*  Allow the target to run free, until it either hits a
                      breakpoint or accesses an address watched by hardware. */
  else    /*  Hardware resources exceeded; must watch via software. */
    if  ( $list_length ($machine_step_watchlist) > 0 )
      $step ( "into", "machine_instruction_granularity" );
    else if  ( $list_length ($source_step_watchlist) > 0 )
      $step ( "into", "source_line_granularity" );
    else
      $step ( "through", "source_line_granularity" );
    /*  Always awaken any and all watchers at the machine instruction level: */
    $awaken_watch_list ($machine_step_watchlist);

    if  ( $source_line_boundary($PC) ) {
      /*  If currently positioned at beginning of a new source line,
          then awaken any and all watchers at the source line level  */
      $awaken_watch_list ($source_step_watchlist);
      /*  Conditionally awaken those watchers that prefer to step
          "through".  Such watches are only awakened if the
          target's current frame matches that at which the watch originated  */
      $awaken_some_on_watch_list ($source_next_watchlist);
    }
}
```

Figure 5: The "top-level" alternative to loosely-bound watch-loops.

[4] "DBX (1)" in *UNIX Programmer's Manual*, 4.2 Berkeley System Distribution, Vol. 1, Computer Science Division, University of California, Berkeley, CA, August 1983.

[5] D. R. Hanson, "Event associations in SNOBOL4 for program debugging," *SOFTWARE—Practice and Experience*, Volume 8, Number 2, pp. 115-129, March-April 1978.

[6] M. S. Johnson, *The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment*, Ph.D. Dissertation, The University of British Columbia, TR-78-6, August 1978.

[7] H. P. Katseff, "Sdb: a symbolic debugger," in *UNIX Programmer's Manual, 7th Edition, Vol. 2C*, Bell Laboratories, Holmdel, NJ, January 1979.

[8] S. Kaufer, R. Lopez, and S. Pratap, "Saber-C, an interpreter-based programming environment for the C language," *Proc. Summer 1988 USENIX Conference*, pp. 161-171, June 1988.

[9] T. J. Killian, "Processes as files," *Proc. Summer 1984 USENIX Conference*, pp. 203-207, June 1984.

[10] C.E. McDowell and D.P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, Volume 21, Number 4, pp. 593-622, December 1989.

[11] R.A. Olsson, R.H. Crawford, and W.W. Ho, "A dataflow approach to event-based debugging," *SOFTWARE—Practice and Experience*, Volume 21, Number 2, pages 209-229, February 1991.

[12] R.A. Olsson, R.H. Crawford, W.W. Ho, and C.E. Wee, "Sequential Debugging at a High Level of Abstraction," *IEEE SOFTWARE 8*, Volume 8, Number 3, pp. 27-36, May 1991.

[13] R.M. Stallman, *GDB Manual (The GNU Source-Level Debugger), Third Edition, GDB version 3.1*, Free Software Foundation, Cambridge, MA, January 1989.

[14] *VMS Debugger Manual, AA-LA59A-TE, VMS version 5.0*, Digital Equipment Company, Maynard, MA, April 1988.