Acid: A Debugger Built From A Language

Phil Winterbottom

Lucent Technologies Inc

ABSTRACT

Acid is an unusual source-level symbolic debugger for Plan 9. It is implemented as a language interpreter with specialized primitives that provide debugger support. Programs written in the language manipulate one or more target processes; variables in the language represent the symbols, state, and resources of those processes. This structure allows complex interaction between the debugger and the target program and provides a convenient method of parameterizing differences between machine architectures. Although some effort is required to learn the debugging language, the richness and flexibility of the debugging environment encourages new ways of reasoning about the way programs run and the conditions under which they fail.

1. Introduction

The size and complexity of programs have increased in proportion to processor speed and memory but the interface between debugger and programmer has changed little. Graphical user interfaces have eased some of the tedious aspects of the interaction. A graphical interface is a convenient means for navigating through source and data structures but provides little benefit for process control. The introduction of a new concurrent language, Alef [Win93], emphasized the inadequacies of the existing Plan 9 [Pike90] debugger *db*, a distant relative of *adb*, and made it clear that a new debugger was required.

Current debuggers like *dbx*, *sdb*, and *gdb* are limited to answering only the questions their authors envisage. As a result, they supply a plethora of specialized commands, each attempting to anticipate a specific question a user may ask. When a debugging situation arises that is beyond the scope of the command set, the tool is useless. Further, it is often tedious or impossible to reproduce an anomalous state of the program, especially when the state is embedded in the program's data structures.

Acid applies some ideas found in CAD software used for hardware test and simulation. It is based on the notion that the state and resources of a program are best represented and manipulated by a language. The state and resources, such as memory, registers, variables, type information and source code are represented by variables in the language. Expressions provide a computation mechanism and control statements allow repetitive or selective interpretation based on the result of expression evaluation. The heart of the Acid debugger is an interpreter for a small typeless language whose operators mirror the operations of C and Alef, which in turn correspond well to the basic operations of the machine. The interpreter itself knows nothing of the underlying hardware; it deals with the program state and resources in the abstract. Fundamental routines to control processes, read files, and interface to the system are implemented as builtin functions available to the interpreter. The actual debugger functionality is coded in Acid; commands are implemented as Acid functions.

This language-based approach has several advantages. Most importantly, programs written in Acid, including most of the debugger itself, are inherently portable. Furthermore, Acid avoids the limitations other debuggers impose when debugging parallel programs. Instead of embedding a fixed process model in the debugger, Acid allows the programmer to adapt the debugger to handle an arbitrary process partitioning or program structure. The ability to interact dynamically with an executing process provides clear advantages over debuggers constrained to probe a static image. Finally, the Acid language is a powerful vehicle for expressing assertions about logic, process state, and the contents of data structures. When combined with dynamic interaction it allows a limited form of automated program verification without requiring modification or recompilation of the source code. The language is also an excellent vehicle for preserving a test suite for later regression testing.

Originally appeared in *Proc. of the Winter 1994 USENIX Conf.*, pp. 211-222, San Francisco, CA; and subsequently in the *Plan 9 Programmer's Manual, Volume 2 (Second Edition)*.

The debugger may be customized by its users; standard functions may be modified or extended to suit a particular application or preference. For example, the kernel developers in our group require a command set supporting assembler-level debugging while the application programmers prefer source-level functionality. Although the default library is biased toward assembler-level debugging, it is easily modified to provide a convenient source-level interface. The debugger itself does not change; the user combines primitives and existing Acid functions in different ways to implement the desired interface.

2. Related Work

DUEL [Gol93], an extension to *gdb* [Stal91], proposes using a high level expression evaluator to solve some of these problems. The evaluator provides iterators to loop over data structures and conditionals to control evaluation of expressions. The author shows that complex state queries can be formulated by combining concise expressions but this only addresses part of the problem. A program is a dynamic entity; questions asked when the program is in a static state are meaningful only after the program has been 'caught' in that state. The framework for manipulating the program is still as primitive as the underlying debugger. While DUEL provides a means to probe data structures it entirely neglects the most beneficial aspect of debugging languages: the ability to control processes. Acid is structured around a thread of control that passes between the interpreter and the target program.

The NeD debugger [May92] is a set of extensions to TCL [Ous90] that provide debugging primitives. The resulting language, NeDtcl, is used to implement a portable interface between a conventional debugger, pdb [May90], and a server that executes NeDtcl programs operating on the target program. Execution of the NeDtcl programs implements the debugging primitives that pdb expects. NeD is targeted at multi-process debugging across a network, and proves the flexibility of a language as a means of communication between debugging tools. Whereas NeD provides an interface between a conventional debugger and the process it debugs, Acid is the debugger itself. While NeD has some of the ideas found in Acid it is targeted toward a different purpose. Acid seeks to integrate the manipulation of a program's resources into the debugger while NeD provides a flexible interconnect between components of the debugging environment. The choice of TCL is appropriate for its use in NeD but is not suitable for Acid. Acid relies on the coupling of the type system with expression evaluation, which are the root of its design, to provide the debugging primitives.

Dalek [Ols90] is an event based language extension to gdb. State transitions in the target program cause events to be queued for processing by the debugging language.

Acid has many of the advantages of same process or *local agent* debuggers, like Parasight [Aral], without the need for dynamic linking or shared memory. Acid improves on the ideas of these other systems by completely integrating all aspects of the debugging process into the language environment. Of particular importance is the relationship between Acid variables, program symbols, source code, registers and type information. This integration is made possible by the design of the Acid language.

Interpreted languages such as Lisp and Smalltalk are able to provide richer debugging environments through more complete information than their compiled counterparts. Acid is a means to gather and represent similar information about compiled programs through cooperation with the compilation tools and library implementers.

3. Acid the Language

Acid is a small interpreted language targeted to its debugging task. It focuses on representing program state and addressing data rather than expressing complex computations. Program state is *addressable* from an Acid program. In addition to parsing and executing expressions and providing an architecture-independent interface to the target process, the interpreter supplies a mark-and-scan garbage collector to manage storage.

Every Acid session begins with the loading of the Acid libraries. These libraries contain functions, written in Acid, that provide a standard debugging environment including breakpoint management, stepping by instruction or statement, stack tracing, and access to variables, memory, and registers. The library contains 600 lines of Acid code and provides functionality similar to *dbx*. Following the loading of the system library, Acid loads user-specified libraries; this load sequence allows the user to augment or override the standard commands to customize the debugging environment. When all libraries are loaded, Acid issues an interactive prompt and begins evaluating expressions entered by the user. The Acid 'commands' are actually invocations of builtin primitives or previously defined Acid functions. Acid evaluates each expression as it is entered and prints the result.

4. Types and Variables

Acid variables are of four basic types: *integer*, *string*, *float*, and *list*. The type of a variable is inferred by the type of the right-hand side of an assignment expression. Many of the operators can be applied to more than one type; for these operators the action of the operator is determined by the type of its operands. For example, the + operator adds *integer* and *float* operands, and concatenates *string* and *list* operands. Lists are the only complex type in Acid; there are no arrays, structures or pointers. Operators provide `head`, `tail`, `append` and `delete` operations. Lists can also be indexed like arrays.

Acid has two levels of scope: global and local. Function parameters and variables declared in a function body using the `local` keyword are created at entry to the function and exist for the lifetime of a function. Global variables are created by assignment and need not be declared. All variables and functions in the program being debugged are entered in the Acid symbol table as global variables during Acid initialization. Conflicting variable names are resolved by prefixing enough '$' characters to make them unique. Syntactically, Acid variables and target program symbols are referenced identically. However, the variables are managed differently in the Acid symbol table and the user must be aware of this distinction. The value of an Acid variable is stored in the symbol table; a reference returns the value. The symbol table entry for a variable or function in the target program contains the address of that symbol in the image of the program. Thus, the value of a program variable is accessed by indirect reference through the Acid variable that has the same name; the value of an Acid variable is the address of the corresponding program variable.

5. Control Flow

The `while` and `loop` statements implement looping. The former is similar to the same statement in C. The latter evaluates starting and ending expressions yielding integers and iterates while an incrementing loop index is within the bounds of those expressions.

```
acid: i = 0; loop 1,5 do print(i=i+1)
0x00000001
0x00000002
0x00000003
0x00000004
0x00000005
acid:
```

The traditional `if-then-else` statement implements conditional execution.

6. Addressing

Two indirection operators allow Acid to access values in the program being debugged. The `*` operator fetches a value from the memory image of an executing process; the `@` operator fetches a value from the text file of the process. When either operator appears on the left side of an assignment, the value is written rather than read.

The indirection operator must know the size of the object referenced by a variable. The Plan 9 compilers neglect to include this information in the program symbol table, so Acid cannot derive this information implicitly. Instead Acid variables have formats. The format is a code letter specifying the printing style and the effect of some of the operators on that variable. The indirection operators look at the format code to determine the number of bytes to read or write. The format codes are derived from the format letters used by *db*. By default, symbol table variables and numeric constants are assigned the format code `'X'` which specifies 32-bit hexadecimal. Printing such a variable yields output of the form `0x00123456`. An indirect reference through the variable fetches 32 bits of data at the address indicated by the variable. Other formats specify various data types, for example `i` an instruction, `D` a signed 32 bit decimal, `s` a null-terminated string. The `fmt` function allows the user to change the format code of a variable to control the printing format and operator side effects. This function evaluates the expression supplied as the first argument, attaches the format code supplied as the second argument to the result and returns that value. If the result is assigned to a variable, the new format code applies to that variable. For convenience, Acid provides the \ operator as a shorthand infix form of `fmt`. For example:

```
acid: x=10
acid: x              // print x in hex
0x0000000a
acid: x = fmt(x, 'D')        // make x type decimal
acid: print(x, fmt(x, 'X'), x\X) // print x in decimal & hex
10 0x0000000a 0x0000000a
acid: x              // print x in decimal
10
acid: x\o            // print x in octal
000000000012
```

The ++ and -- operators increment or decrement a variable by an amount determined by its format code. Some formats imply a non-fixed size. For example, the `i` format code disassembles an instruction into a string. On a 68020, which has variable length instructions:

```
acid: p=main\i                     // p=addr(main), type INST
acid: loop 1,5 do print(p\X, @p++) // disassemble 5 instr's
0x0000222e LEA  0xffffe948(A7),A7
0x00002232 MOVL s+0x4(A7),A2
0x00002236 PEA  0x2f($0)
0x0000223a MOVL A2,-(A7)
0x0000223c BSR  utfrrune
acid:
```

Here, `main` is the address of the function of the same name in the program under test. The loop retrieves the five instructions beginning at that address and then prints the address and the assembly language representation of each. Notice that the stride of the increment operator varies with the size of the instruction: the `MOVL` at `0x0000223a` is a two byte instruction while all others are four bytes long.

Registers are treated as normal program variables referenced by their symbolic assembler language names. When a process stops, the register set is saved by the kernel at a known virtual address in the process memory map. The Acid variables associated with the registers point to the saved values and the `*` indirection operator can then be used to read and write the register set. Since the registers are accessed via Acid variables they may be used in arbitrary expressions.

```
acid: PC                            // addr of saved PC
0xc0000f60
acid: *PC
0x0000623c                          // contents of PC
acid: *PC\a
main
acid: *R1=10                        // modify R1
acid: asm(*PC+4)                    // disassemble @ PC+4
main+0x4 0x00006240     MOVW    R31,0x0(R29)
main+0x8 0x00006244     MOVW    $setR30(SB),R30
main+0x10 0x0000624c    MOVW    R1,_clock(SB)
```

Here, the saved `PC` is stored at address `0xc0000f60`; its current content is `0x0000623c`. The 'a' format code converts this value to a string specifying the address as an offset beyond the nearest symbol. After setting the value of register `1`, the example uses the `asm` command to disassemble a short section of code beginning at four bytes beyond the current value of the `PC`.

7. Process Interface

A program executing under Acid is monitored through the *proc* file system interface provided by Plan 9. Textual messages written to the `ctl` file control the execution of the process. For example writing `waitstop` to the control file causes the write to block until the target process enters the kernel and is stopped. When the process is stopped the write completes. The `startstop` message starts the target process and then does a `waitstop` action. Synchronization between the debugger and the target process is determined by the actions of the various messages. Some operate asynchronously to the target process and always complete immediately, others block until the action completes. The asynchronous messages allow Acid to control several processes simultaneously.

The interpreter has builtin functions named after each of the control messages. The functions take a process id as argument. Any time a control message causes the program to execute instructions the interpreter performs two actions when the control operation has completed. The Acid variables pointing at the register set are fixed up to point at the saved registers, and then the user defined function `stopped` is executed. The `stopped` function may print the current address, line of source or instruction and return to interactive mode. Alternatively it may traverse a complex data structure, gather statistics and then set the program running again.

Several Acid variables are maintained by the debugger rather than the programmer. These variables allow generic Acid code to deal with the current process, architecture specifics or the symbol table. The variable `pid` is the process id of the current process Acid is debugging. The variable `symbols` contains a list of lists where each sublist contains the symbol name, its type and the value of the symbol. The variable `registers` contains a list of the machine-specific register names. Global symbols in the target program can be referenced directly by name from Acid. Local variables are referenced using the colon operator as `function:variable`.

8. Source Level Debugging

Acid provides several builtin functions to manipulate source code. The `file` function reads a text file, inserting each line into a list. The `pcfile` and `pcline` functions each take an address as an argument. The first returns a string containing the name of the source file and the second returns an integer containing the line number of the source line containing the instruction at the address.

```
acid: pcfile(main)      // file containing main
main.c
acid: pcline(main)      // line # of main in source
11
acid: file(pcfile(main))[pcline(main)]  // print that line
main(int argc, char *argv[])
acid: src(*PC)          // print statements nearby
 9
 10 void
>11 main(int argc, char *argv[])
 12 {
 13 int a;
```

In this example, the three primitives are combined in an expression to print a line of source code associated with an address. The `src` function prints a few lines of source around the address supplied as its argument. A companion routine, `Bsrc`, communicates with the external editor `sam`. Given an address, it loads the corresponding source file into the editor and highlights the line containing the address. This simple interface is easily extended to more complex functions. For example, the `step` function can select the current file and line in the editor each time the target program stops, giving the user a visual trace of the execution path of the program. A more complete interface allowing two way communication between Acid and the `acme` user interface [Pike93] is under construction. A filter between the debugger and the user interface provides interpretation of results from both sides of the interface. This allows the programming environment to interact with the debugger and vice-versa, a capability missing from the `sam` interface. The `src` and `Bsrc` functions are both written in Acid code using the file and line primitives. Acid provides library functions to step through source level statements and functions. Furthermore, addresses in Acid expressions can be specified by source file and line. Source code is manipulated in the Acid *list* data type.

9. The Acid Library

The following examples define some useful commands and illustrate the interaction of the debugger and the interpreter.

```
defn bpset(addr)                          // set breakpoint
{
    if match(addr, bplist) >= 0 then
        print("bkpoint already set:", addr\a, "\n");
    else {
        *fmt(addr, bpfmt) = bpinst;   // plant it
        bplist = append bplist, addr; // add to list
    }
}
```

The `bpset` function plants a break point in memory. The function starts by using the `match` builtin to search the breakpoint list to determine if a breakpoint is already set at the address. The indirection operator, controlled by the format code returned by the `fmt` primitive, is used to plant the breakpoint in memory. The variables `bpfmt` and `bpinst` are Acid global variables containing the format code specifying the size of the breakpoint instruction and the breakpoint instruction itself. These variables are set by architecture-dependent library code when the debugger first attaches to the executing image. Finally the address of the breakpoint is appended to the breakpoint list, `bplist`.

```
defn step()             // single step
{
    local lst, lpl, addr, bput;

    bput = 0;           // sitting on bkpoint
    if match(*PC, bplist) >= 0 then {
        bput = fmt(*PC, bpfmt); // save current addr
        *bput = @bput;      // replace it
    }

    lst = follow(*PC);      // get follow set

    lpl = lst;
    while lpl do {          // place breakpoints
        *(head lpl) = bpinst;
        lpl = tail lpl;
    }

    startstop(pid);         // do the step

    while lst do {          // remove breakpoints
        addr = fmt(head lst, bpfmt);
        *addr = @addr;      // replace instr.
        lst = tail lst;
    }
    if bput != 0 then
        *bput = bpinst;     // restore breakpoint
}
```

The `step` function executes a single assembler instruction. If the `PC` is sitting on a breakpoint, the address and size of the breakpoint are saved. The breakpoint instruction is then removed using the `@` operator to fetch `bpfmt` bytes from the text file and to place it into the memory of the executing process using the `*` operator. The `follow` function is an Acid builtin which returns a follow-set: a list of instruction addresses which could be executed next. If the instruction stored at the `PC` is a branch instruction, the list contains the addresses of the next instruction and the branch destination; otherwise, it contains only the address of the next instruction. The follow-set is then used to replace each possible following instruction with a breakpoint instruction. The original instructions need not be saved; they remain in their unaltered state in the text file. The `startstop` builtin writes the 'startstop' message to the *proc* control file for the process named `pid`. The target process executes until some condition causes it to enter the kernel, in this case, the execution of a breakpoint. When the process blocks, the debugger regains control and invokes the Acid library function `stopped` which reports the address and cause of the blockage. The `startstop` function completes and returns to the `step` function where the follow-set is used to replace the breakpoints placed earlier. Finally, if the address of the original `PC` contained a breakpoint, it is replaced.

Notice that this approach to process control is inherently portable; the Acid code is shared by the debuggers for all architectures. Acid variables and builtin functions provide a transparent interface to architecture-dependent values and functions. Here the breakpoint value and format are referenced through Acid variables and the `follow` primitive masks the differences in the underlying instruction set.

The `next` function, similar to the *dbx* command of the same name, is a simpler example. This function steps through a single source statement but steps over function calls.

```
defn next()
{
    local sp, bound;

    sp = *SP;           // save starting SP
    bound = fnbound(*PC);       // begin & end of fn.
    stmnt();            // step 1 statement
    pc = *PC;
    if pc >= bound[0] && pc < bound[1] then
        return {};

    while (pc<bound[0] || pc>bound[1]) && sp>=*SP do {
        step();
        pc = *PC;
    }
    src(*PC);
}
```

The `next` function starts by saving the current stack pointer in a local variable. It then uses the Acid library function `fnbound` to return the addresses of the first and last instructions in the current function in a list. The `stmnt` function executes a single source statement and then uses `src` to print a few lines of source around the new `PC`. If the new value of the `PC` remains in the current function, `next` returns. When the executed statement is a function call or a return from a function, the new value of the `PC` is outside the bounds calculated by `fnbound` and the test of the `while` loop is evaluated. If the statement was a return, the new value of the stack pointer is greater than the original value and the loop completes without execution. Otherwise, the loop is entered and instructions are continually executed until the value of the `PC` is between the bounds calculated earlier. At that point, execution ceases and a few lines of source in the vicinity of the `PC` are printed.

Acid provides concise and elegant expression for control and manipulation of target programs. These examples demonstrate how a few well-chosen primitives can be combined to create a rich debugging environment.

10. Dealing With Multiple Architectures

A single binary of Acid may be used to debug a program running on any of the five processor architectures supported by Plan 9. For example, Plan 9 allows a user on a MIPS to import the *proc* file system from an i486-based PC and remotely debug a program executing on that processor.

Two levels of abstraction provide this architecture independence. On the lowest level, a Plan 9 library supplies functions to decode the file header of the program being debugged and select a table of system parameters and a jump vector of architecture-dependent functions based on the magic number. Among these functions are byte-order-independent access to memory and text files, stack manipulation, disassembly, and floating point number interpretation. The second level of abstraction is supplied by Acid. It consists of primitives and approximately 200 lines of architecture-dependent Acid library code that interface the interpreter to the architecture-dependent library. This layer performs functions such as mapping register names to memory locations, supplying breakpoint values and sizes, and converting processor specific data to Acid data types. An example of the latter is the stack trace function `strace`, which uses the stack traversal functions in the architecture-dependent library to construct a list of lists describing the context of a process. The first level of list selects each function in the trace; subordinate lists contain the names and values of parameters and local variables of the functions. Acid commands and library functions that manipulate and display process state information operate on the list representation and are independent of the underlying architecture.

11. Alef Runtime

Alef is a concurrent programming language, designed specifically for systems programming, which supports both shared variable and message passing paradigms. Alef borrows the C expression syntax but implements a substantially different type system. The language provides a rich set of exception handling, process management, and synchronization primitives, which rely on a runtime system. Alef program bugs are often deadlocks, synchronization failures, or non-termination caused by locks being held incorrectly. In such cases, a process stalls deep in the runtime code and it is clearly unreasonable to expect a programmer using the language to understand the detailed internal semantics of the runtime support functions.

Instead, there is an Alef support library, coded in Acid, that allows the programmer to interpret the program state in terms of Alef operations. Consider the example of a multi-process program stalling because of improper synchronization. A stack trace of the program indicates that it is waiting for an event in some obscure Alef runtime synchronization function. The function itself is irrelevant to the programmer; of greater importance is the identity of the unfulfilled event. Commands in the Alef support library decode the runtime data structures and program state to report the cause of the blockage in terms of the high-level operations available to the Alef programmer. Here, the Acid language acts as a communications medium between Alef implementer and Alef user.

12. Parallel Debugging

The central issue in parallel debugging is how the debugger is multiplexed between the processes comprising the program. Acid has no intrinsic model of process partitioning; it only assumes that parallel programs share a symbol table, though they need not share memory. The `setproc` primitive attaches the debugger to a running process associated with the process ID supplied as its argument and assigns that value to the global variable `pid`, thereby allowing simple rotation among a group of processes. Further, the stack trace primitive is driven by parameters specifying a unique process context, so it is possible to examine the state of cooperating processes without switching the debugger focus from the process of interest. Since Acid is inherently extensible and capable of dynamic interaction with subordinate processes, the programmer can define Acid commands to detect and control complex interactions between processes. In short, the programmer is free to specify how the debugger reacts to events generated in specific threads of the program.

The support for parallel debugging in Acid depends on a crucial kernel modification: when the text segment of a program is written (usually to place a breakpoint), the segment is cloned to prevent other threads from encountering the breakpoint. Although this incurs a slight performance penalty, it is of little importance while debugging.

13. Communication Between Tools

The Plan 9 Alef and C compilers do not embed detailed type information in the symbol table of an executable file. However, they do accept a command line option causing them to emit descriptions of complex data types (e.g., aggregates and abstract data types) to an auxiliary file. The vehicle for expressing this information is Acid source code. When an Acid debugging session is subsequently started, that file is loaded with the other Acid libraries.

For each complex object in the program the compiler generates three pieces of Acid code. The first is a table describing the size and offset of each member of the complex data type. Following is an Acid function, named the same as the object, that formats and prints each member. Finally, Acid declarations associate the Alef or C program variables of a type with the functions to print them. The three forms of declaration are shown in the following example:

```
struct Bitmap {
    Rectangle    0 r;
    Rectangle   16 clipr;
    'D'   32 ldepth;
    'D'   36 id;
    'X'   40 cache;
};
```

```
defn
Bitmap(addr) {
    complex Bitmap addr;
    print("Rectangle r {\n");
    Rectangle(addr.r);
    print("}\n");
    print("Rectangle clipr {\n");
    Rectangle(addr.clipr);
    print("}\n");
    print(" ldepth  ", addr.ldepth, "\n");
    print(" id  ", addr.id, "\n");
    print(" cache   ", addr.cache, "\n");
};

complex Bitmap darkgrey;
complex Bitmap Window_settag:b;
```

The `struct` declaration specifies decoding instructions for the complex type named `Bitmap`. Although the syntax is superficially similar to a C structure declaration, the semantics differ markedly: the C declaration specifies a layout, while the Acid declaration tells how to decode it. The declaration specifies a type, an offset, and name for each member of the complex object. The type is either the name of another complex declaration, for example, `Rectangle`, or a format code. The offset is the number of bytes from the start of the object to the member and the name is the member's name in the Alef or C declaration. This type description is a close match for C and Alef, but is simple enough to be language independent.

The `Bitmap` function expects the address of a `Bitmap` as its only argument. It uses the decoding information contained in the `Bitmap` structure declaration to extract, format, and print the value of each member of the complex object pointed to by the argument. The Alef compiler emits code to call other Acid functions where a member is another complex type; here, `Bitmap` calls `Rectangle` to print its contents.

The `complex` declarations associate Alef variables with complex types. In the example, `darkgrey` is the name of a global variable of type `Bitmap` in the program being debugged. Whenever the name `darkgrey` is evaluated by Acid, it automatically calls the `Bitmap` function with the address of `darkgrey` as the argument. The second `complex` declaration associates a local variable or parameter named `b` in function `Window_settag` with the `Bitmap` complex data type.

Acid borrows the C operators `.` and `->` to access the decoding parameters of a member of a complex type. Although this representation is sufficiently general for describing the decoding of both C and Alef complex data types, it may prove too restrictive for target languages with more complicated type systems. Further, the assumption that the compiler can select the proper Acid format code for each basic type in the language is somewhat naive. For example, when a member of a complex type is a pointer, it is assigned a hexadecimal type code; integer members are always assigned a decimal type code. This heuristic proves inaccurate when an integer field is a bit mask or set of bit flags which are more appropriately displayed in hexadecimal or octal.

14. Code Verification

Acid's ability to interact dynamically with an executing program allows passive test and verification of the target program. For example, a common concern is leak detection in programs using `malloc`. Of interest are two items: finding memory that was allocated but never freed and detecting bad pointers passed to `free`. An auxiliary Acid library contains Acid functions to monitor the execution of a program and detect these faults, either as they happen or in the automated post-mortem analysis of the memory arena. In the following example, the `sort` command is run under the control of the Acid memory leak library.

```
helix% acid -l malloc /bin/sort
/bin/sort: mips plan 9 executable
/lib/acid/port
/lib/acid/mips
/lib/acid/malloc
acid: go()
now
is
the
time
<ctrl-d>
is
now
the
time
27680 : breakpoint  _exits+0x4  MOVW    $0x8,R1
acid:
```

The `go` command creates a process and plants breakpoints at the entry to `malloc` and `free`. The program is then started and continues until it exits or stops. If the reason for stopping is anything other than the breakpoints in `malloc` and `free`, Acid prints the usual status information and returns to the interactive prompt.

When the process stops on entering `malloc`, the debugger must capture and save the address that `malloc` will return. After saving a stack trace so the calling routine can be identified, it places a breakpoint at the return address and restarts the program. When `malloc` returns, the breakpoint stops the program, allowing the debugger to grab the address of the new memory block from the return register. The address and stack trace are added to the list of outstanding memory blocks, the breakpoint is removed from the return point, and the process is restarted.

When the process stops at the beginning of `free`, the memory address supplied as the argument is compared to the list of outstanding memory blocks. If it is not found an error message and a stack trace of the call is reported; otherwise, the address is deleted from the list.

When the program exits, the list of outstanding memory blocks contains the addresses of all blocks that were allocated but never freed. The `leak` library function traverses the list producing a report describing the allocated blocks.

```
acid: leak()
Lost a total of 524288 bytes from:
    malloc() malloc.c:32 called from dofile+0xe8 sort.c:217
    dofile() sort.c:190 called from main+0xac sort.c:161
    main() sort.c:128 called from _main+0x20 main9.s:10
Lost a total of 64 bytes from:
    malloc() malloc.c:32 called from newline+0xfc sort.c:280
    newline() sort.c:248 called from dofile+0x110 sort.c:222
    dofile() sort.c:190 called from main+0xac sort.c:161
    main() sort.c:128 called from _main+0x20 main9.s:10
Lost a total of 64 bytes from:
    malloc() malloc.c:32 called from realloc+0x14 malloc.c:129
    realloc() malloc.c:123 called from bldkey+0x358 sort.c:1388
    buildkey() sort.c:1345 called from newline+0x150 sort.c:285
    newline() sort.c:248 called from dofile+0x110 sort.c:222
    dofile() sort.c:190 called from main+0xac sort.c:161
    main() sort.c:128 called from _main+0x20 main9.s:10
acid: refs()
data...bss...stack...
acid: leak()
acid:
```

The presence of a block in the allocation list does not imply it is there because of a leak; for instance, it may have been in use when the program terminated. The `refs()` library function scans the *data*, *bss*, and *stack* segments of the process looking for pointers into the allocated blocks. When one is found, the block is deleted from the outstanding block list. The `leak` function is used again to report the blocks remaining allocated and unreferenced. This strategy proves effective in detecting disconnected (but non-circular) data structures.

The leak detection process is entirely passive. The program is not specially compiled and the source code is not required. As with the Acid support functions for the Alef runtime environment, the author of the library routines has encapsulated the functionality of the library interface in Acid code. Any programmer may then check a program's use of the library routines without knowledge of either implementation. The performance impact of running leak detection is great (about 10 times slower), but it has not prevented interactive programs like `sam` and the `8½` window system from being tested.

15. Code Coverage

Another common component of software test uses *coverage* analysis. The purpose of the test is to determine which paths through the code have not been executed while running the test suite. This is usually performed by a combination of compiler support and a reporting tool run on the output generated by statements compiled into the program. The compiler emits code that logs the progress of the program as it executes basic blocks and writes the results to a file. The file is then processed by the reporting tool to determine which basic blocks have not been executed.

Acid can perform the same function in a language independent manner without modifying the source, object or binary of the program. The following example shows `ls` being run under the control of the Acid coverage library.

```
philw-helix% acid -l coverage /bin/ls
/bin/ls: mips plan 9 executable
/lib/acid/port
/lib/acid/mips
/lib/acid/coverage
acid: coverage()
acid
newstime
profile
tel
wintool
2: (error) msg: pid=11419 startstop: process exited
acid: analyse(ls)
ls.c:102,105
    102:     return 1;
    103: }
    104: if(db[0].qid.path&CHDIR && dflag==0){
    105:     output();
ls.c:122,126
    122:     memmove(dirbuf+ndir, db, sizeof(Dir));
    123:     dirbuf[ndir].prefix = 0;
    124:     p = utfrrune(s, '/');
    125:     if(p){
    126:         dirbuf[ndir].prefix = s;
```

The `coverage` function begins by looping through the text segment placing breakpoints at the entry to each basic block. The start of each basic block is found using the Acid builtin function `follow`. If the list generated by `follow` contains more than one element, then the addresses mark the start of basic blocks. A breakpoint is placed at each address to detect entry into the block. If the result of `follow` is a single address then no action is taken, and the next address is considered. Acid maintains a list of breakpoints already in place and avoids placing duplicates (an address may be the destination of several branches).

After placing the breakpoints the program is set running. Each time a breakpoint is encountered Acid deletes the address from the breakpoint list, removes the breakpoint from memory and then restarts the program. At any instant the breakpoint list contains the addresses of basic blocks which have not been executed. The `analyse` function reports the lines of source code bounded by basic blocks whose addresses are have not been deleted from the breakpoint list. These are the basic blocks which have not been executed. Program performance is almost unaffected since each breakpoint is executed only once and then removed.

The library contains a total of 128 lines of Acid code. An obvious extension of this algorithm could be used to provide basic block profiling.

16. Conclusion

Acid has two areas of weakness. As with other language-based tools like *awk*, a programmer must learn yet another language to step beyond the normal debugging functions and use the full power of the debugger. Second, the command line interface supplied by the *yacc* parser is inordinately clumsy. Part of the problem relates directly to the use of *yacc* and could be circumvented with a custom parser. However, structural problems would remain: Acid often requires too much typing to execute a simple command. A debugger should prostitute itself to its users, doing whatever is wanted with a minimum of encouragement; commands should be concise and obvious. The language interface is more consistent than an ad hoc command interface but is clumsy to use. Most of these problems are addressed by an Acme interface which is under construction. This should provide the best of both worlds: graphical debugging and access to the underlying acid language when required.

The name space clash between Acid variables, keywords, program variables, and functions is unavoidable. Although it rarely affects a debugging session, it is annoying when it happens and is sometimes difficult to circumvent. The current renaming scheme is too crude; the new names are too hard to remember.

Acid has proved to be a powerful tool whose applications have exceeded expectations. Of its strengths, portability, extensibility and parallel debugging support were by design and provide the expected utility. In retrospect, its use as a tool for code test and verification and as a medium for communicating type information and encapsulating interfaces has provided unanticipated benefits and altered our view of the debugging process.

17. Acknowledgments

Bob Flandrena was the first user and helped prepare the paper. Rob Pike endured three buggy Alef compilers and a new debugger in a single sitting.

18. References

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, ''Plan 9 from Bell Labs'', *UKUUG Proc. of the Summer 1990 Conf.,* London, England, 1990.

[Gol93] M. Golan, D. Hanson, ''DUEL -- A Very High-Level Debugging Language'', *USENIX Proc. of the Winter 1993 Conf.,* San Diego, CA, 1993.

[Lin90] M. A. Linton, ''The Evolution of DBX'', *USENIX Proc. of the Summer 1990 Conf.,* Anaheim, CA, 1990.

[Stal91] R. M. Stallman, R. H. Pesch, ''Using GDB: A guide to the GNU source level debugger'', Technical Report, Free Software Foundation, Cambridge, MA, 1991.

[Win93] P. Winterbottom, ''Alef reference Manual'', reprinted in this volume.

[Pike93] Rob Pike, ''Acme: A User Interface for Programmers'', *USENIX Proc. of the Winter 1994 Conf.,* San Francisco, CA, reprinted in this volume.

[Ols90] Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho, ''Dalek: A GNU, improved programmable debugger'', *USENIX Proc. of the Summer 1990 Conf.,* Anaheim, CA.

[May92] Paul Maybee, ''NeD: The Network Extensible Debugger'' *USENIX Proc. of the Summer 1992 Conf.,* San Antonio, TX.

[Aral] Ziya Aral, Ilya Gertner, and Greg Schaffer, ''Efficient debugging primitives for multiprocessors'', *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* SIGPLAN notices Nr. 22, May 1989.