Reprinted from the Proceedings of the USENIX 1997 Annual Technical Conference, Anaheim, CA, Jan. 1997, pp. 183-174.

A Simple and Extensible Graphical Debugger

David R. Hanson and Jeffrey L. Korn
Department of Computer Science, Princeton University,
35 Olden St., Princeton, NJ 08544
{drh,jlk}@cs.princeton.edu

Abstract

deet is a simple but powerful debugger for ANSI C and Java. It differs from conventional debuggers in that it is machine-independent, graphical, programmable, distributed, extensible, and small. Low-level operations are performed by communicating with a "nub," which is a small set of machine-dependent functions that are embedded in the target program at compile-time, or are implemented on top of existing debuggers. deet has a set of commands that communicate with the target's nub. The target and deet communicate by passing messages through a pipe or socket, so they can be on a different machines. deet is implemented in tksh, an extension of the Korn shell that provides the graphical facilities of Tcl/Tk. Users can browse source files, set breakpoints, watch variables, and examine data structures by pointing and clicking. Additional facilities, like conditional breakpoints, can be written in either Tcl or the shell. Most debuggers are large and complicated, deet is less than 1,500 lines of shell plus a few hundred lines of machine-specific nub code. It is thus easy to understand, modify, and extend. We describe an implementation of the nub API for Java and an implementation that is layered on top of gdb. We have also implemented a version of gdb using the nub API, which demonstrates the modularity of the design.

1 Introduction

Traditional UNIX debuggers are indispensable tools for locating and fixing program errors. Despite their importance and pervasiveness, they continue to harbor inadequacies that limit their usability. For example, UNIX debuggers typically have textual user interfaces that are cryptic at best. When debuggers are hard to use, programmers tend to litter their programs with print statements instead of using a debugger.

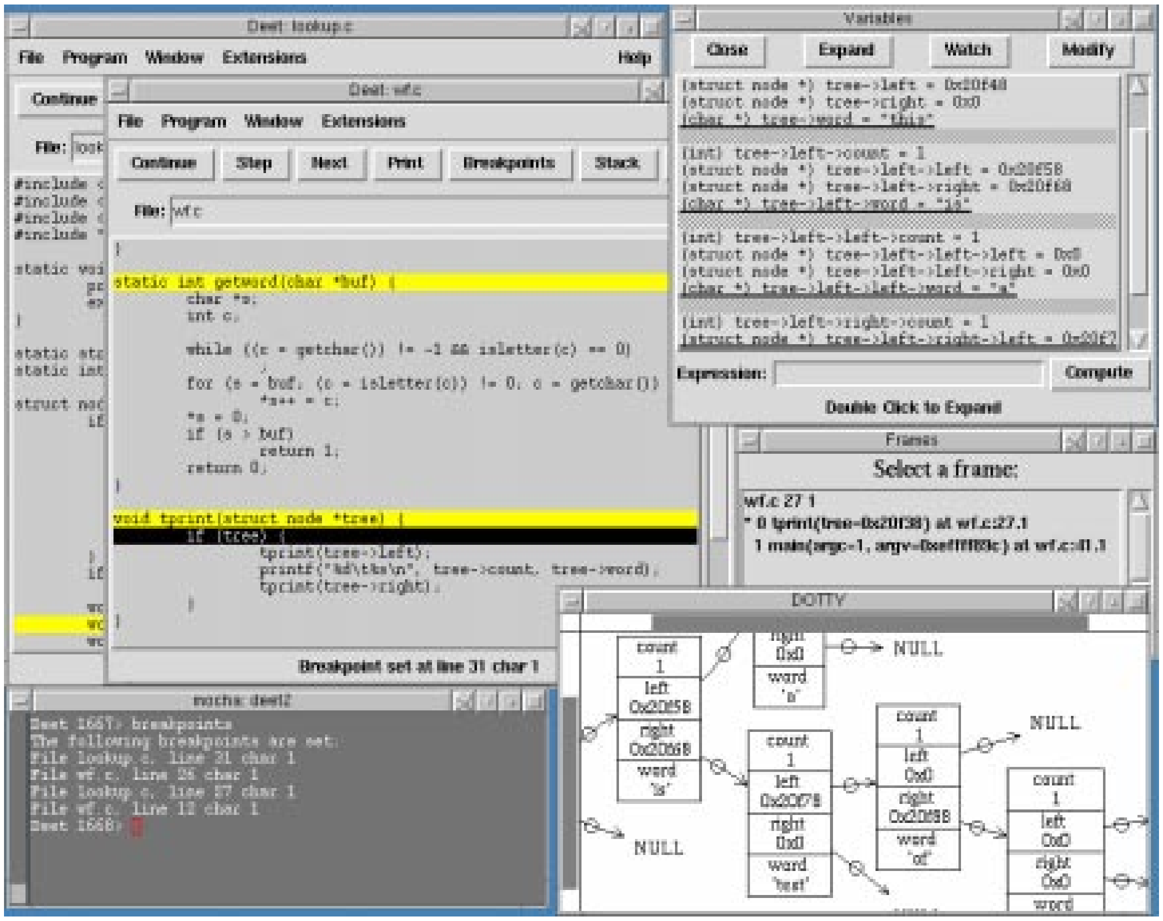
While most PC debuggers run on only one operating system and one architecture, UNIX debuggers must deal with portability issues. Debuggers are notoriously machine-dependent programs; they depend on the target architecture, operating system, compiler, and linker. Thus, porting a debugger from one variant of UNIX to another can require a substantial amount of effort. For example, about one-third of gdb's source code is machine-dependent.

Few debuggers have programming facilities in which, for example, programmers can write application-specific debugging code. Such code is useful for nontrivial queries of data structures, such as displaying the second to last element in a linked list, or all the positive elements in an array. Other examples include setting conditional breakpoints and automating program testing. Debuggers that support programming facilities do exist, but often the language is idiosyncratic to either the debugger or the source language, or both, and hard to learn.

Most debuggers are large and complex programs; for example, gdb [14] is about 150,000 lines of C. This complexity has some unfortunate consequences. First, debuggers are often themselves buggy, because, like any large program, their complexity and size makes them prone to errors and to inconsistent behaviors on different platforms. Second, debuggers are usually difficult to extend, because their implementations may be hard to understand and to modify.

deet (*d*esktop *e*rror *e*limination *t*ool) addresses these shortcomings. It provides both textual and graphical interfaces to make it easy to use. Users can perform most debugging actions by pointing and clicking, and data structures can be displayed graphically. The GUI is written with Tk [12]. deet is also programmable: Its capabilities can be extended by writing in either Tcl or in tksh, a variant of the Korn shell [8].

Nearly all of deet's implementation is machine-independent. It uses a small "nub" that provides facilities for communicating with the debugger and controlling the target. The nub-based approach permits deet to debug a target running on another machine. Figure 1 shows the screen of a typical debugging session. deet doesn't attempt to match debuggers like gdb feature-for-feature; for example, deet can't examine core dumps, evaluate arbitrary C expressions, or debug at the assembly-language level. Nevertheless, its implementation is sur-



```
}

static int getword(char *buf) {
        char *s;
        int c;

        while ((c = getchar()) != -1 && isletter(c) == 0)
                ;
        for (s = buf; (c = isletter(c)) != 0; c = getchar())
                *s++ = c;
        *s = 0;
        if (s > buf)
                return 1;
        return 0;
}

void tprint(struct node *tree) {
        if (tree) {
                tprint(tree->left);
                printf("%d\t%s\n", tree->count, tree->word);
                tprint(tree->right);
        }
}
```

Breakpoint set at line 31 char 1

```
(struct node *) tree->left = 0x20f48
(struct node *) tree->right = 0x0
(char *) tree->word = "this"

(int) tree->left->count = 1
(struct node *) tree->left->left = 0x20f58
(struct node *) tree->left->right = 0x20f68
(char *) tree->left->word = "is"

(int) tree->left->left->count = 1
(struct node *) tree->left->left->left = 0x0
(struct node *) tree->left->left->right = 0x0
(char *) tree->left->left->word = "a"

(int) tree->left->right->count = 1
(struct node *) tree->left->right->left = 0x20f2
```

```
wf.c 27 1
* 0 tprint(tree=0x20f38) at wf.c:27.1
  1 main(argc=1, argv=0xeffff83c) at wf.c:41.1
```

```
Deet 1667> breakpoints
The following breakpoints are set:
File lookup.c, line 21 char 1
File wf.c, line 26 char 1
File lookup.c, line 37 char 1
File wf.c, line 12 char 1
Deet 1668>
```

Figure 1: `deet` screen dump

prisingly simple. Its complete source is approximately 2,500 lines of shell and C.

2 Using `deet`

`deet`'s features are best explained by seeing it in action. First, the target program is compiled by `lcc` [3] with the appropriate debugging option to embed the nub in the target:

```
$ lcc -Wf-g4 wf.c lookup.c
```

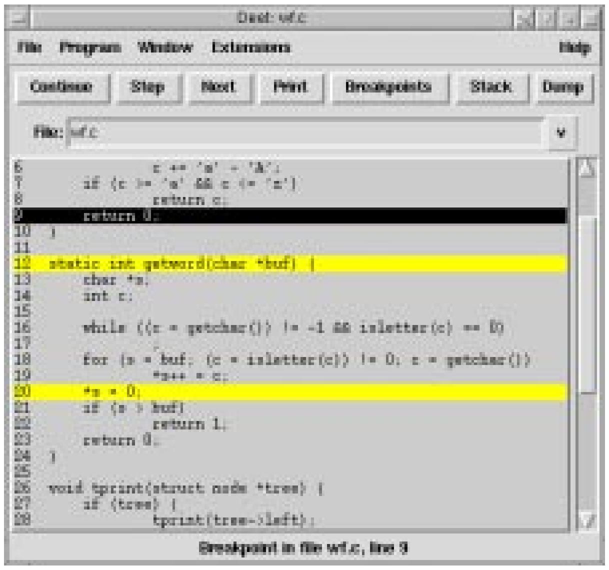
Here and in the displays below, *`slanted`* type identifies user input. When the generated `a.out` is executed, the debugger specified by the environment variable `DEBUGGER` is also started, so

```
$ DEBUGGER=deet a.out
```

starts both `a.out` and `deet`. At this point, the source window shown in Figure 2 appears. The user is prompted for textual `deet` commands in the shell window from which `a.out` was invoked, but most debugging actions are performed with the mouse.

Single-clicking on a line highlights that line if it contains a breakpoint; double-clicking on the line sets the breakpoint. `deet` can set breakpoints on expressions, not just statements, so there may be more than one breakpoint in a line. When a line has multiple breakpoints, double-clicking sets the breakpoint closest to the cursor. Breakpoints are indicated in the window in lighter shading or in yellow (see Figure 2). Double-clicking on a breakpoint that has already been set removes the breakpoint.

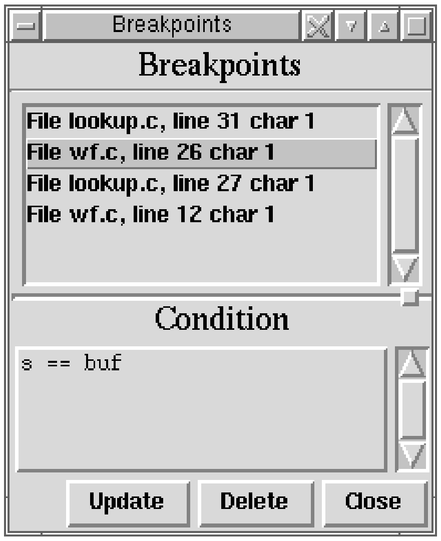
The breakpoints window, like the one shown in Figure 3, displays a list of all breakpoints and related information about each breakpoint, such as its location and break condition. These conditions are `deet` expressions that are evaluated whenever the breakpoint is reached; if the condition is true, the target stops. When the target stops at a breakpoint, the current source window shows the file and line number of the breakpoint, and reverse video highlights the line containing the breakpoint.



```
6               c += 'a' - 'A';
7        if (c >= 'a' && c <= 'z')
8               return c;
9        return 0;
10   }
11
12   static int getword(char *buf) {
13       char *s;
14       int c;
15
16       while ((c = getchar()) != -1 && isletter(c) == 0)
17
18       for (s = buf; (c = isletter(c)) != 0; c = getchar())
19               *s++ = c;
20       *s = 0;
21       if (s > buf)
22               return 1;
23       return 0;
24   }
25
26   void tprint(struct node *tree) {
27       if (tree) {
28               tprint(tree->left);
```

Breakpoint in file wf.c, line 9

Figure 2: Source window



File lookup.c, line 31 char 1
File wf.c, line 26 char 1
File lookup.c, line 27 char 1
File wf.c, line 12 char 1

s == buf

Figure 3: Breakpoints window

A condition can be changed by highlighting the breakpoint in the breakpoints window and editing the condition field, and a breakpoint can also be removed by clicking the "Delete" button in this window.

The stack can be shown by clicking on the "Stack" button in the source window (see Figure 2). This displays a new window that shows each frame on the stack, from the top down, as illustrated in the right middle portion of Figure 1. An individual frame can be selected, and clicking a button in the stack window performs the corresponding action on that frame. For instance, by clicking the "Dump" button, the names and values of the parameters and locals for that frame are displayed. Clicking the "OK" button causes the source window to display the file and line number of the call to the selected frame.

Highlighting a variable in the source window and clicking "Print" causes a pop-up window to display the value of that variable (see the upper right corner of Figure 2). If the variable is a pointer, a structure, or a union, double-clicking on the variable expands its value. For pointers, the value of the referent is displayed; for structures and unions, the values of the fields are displayed. `deet` also displays the values of variables in balloon help pop-up windows when the cursor is left on top of the variables for sufficient time, similar to Microsoft's Visual C++ debugger [11].

A variable can be modified by clicking "Modify" in the variable window, which prompts the user to enter a new value. A variable may also be watched, which causes its value to be displayed in the variable window and updated as execution passes each potential breakpoint.

Commands may be typed at the debugger in the shell window in a manner similar to `gdb`. For instance, the `breakpoints` command displays the current breakpoints:

```
deet> breakpoints
The following breakpoints are set:
File test/wf.c, line 4 char 28
File test/lookup.c, line 14 char 50
```

Commands are just `tksh` commands, so shell commands like `history`, `pwd`, and `make` can be entered as well.

Most of the state in a `deet` debugging session can be saved and restored later in a subsequent, separate debugging session. This state includes breakpoints and their conditions, locations of files, and user-defined `tksh` functions. `deet` saves the state by writing a shell script that can be interpreted to restore the state.

3 Design

`deet` divides cleanly into two parts: One part interacts with the programmer, and the other part interacts with the target program. The user-interface part is written in `tksh`, a version of the new Korn shell [2, 7] that has been extended to support Tcl [12]. The target program is controlled by a nub, which provides debugging primitives, as detailed below. `deet`'s implementation of and interaction with the nub is also written in `tksh`. Thus, programmers can modify and extend *both* parts of `deet` by writing `tksh` code.

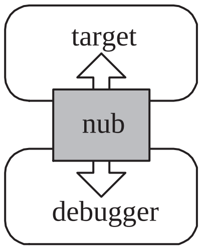


Figure 4: cdb's design

3.1 Cdb and deet

deet is based on cdb [6]. cdb is a machine independent debugger that eliminates machine dependencies by adding a small amount of information into the target program at compile time. cdb communicates with the target through a nub—a small machine-independent interface that constitutes the core functionality of the debugger, as suggested by Figure 4. The nub implementation can be made machine-independent, as cdb shows, but machine-dependent implementations are also possible and are undoubtedly more efficient. The nub is small enough that re-implementing it for new platforms is nearly as easy as porting the machine-independent implementation.

There are four components of cdb:

1. The nub interface, which stands between the debugger and machine-dependent target manipulations.

2. The nub implementation, which consists of the nub interface functions, special code emitted by the compiler to support the nub, and a wrapper around the linker to load the nub and the machine-independent symbol table.

3. A machine-independent symbol table format, which is emitted by the compiler and linked into the target.

4. A simple, text-based debugger that uses the nub to provide minimal functionality; this debugger is intended to be replaced with more sophisticated debuggers, like deet.

Any of the last three components can be replaced with alternative implementations. For instance, the nub can be replaced with a machine-dependent implementation that uses the ptrace system call like most UNIX-specific debuggers, or by one that is layered on top of gdb. The machine-independent symbol tables could be replaced with the usual machine-dependent "stab" symbol tables embedded in UNIX executables. Finally, the debugger itself could be replaced with any program that uses the nub interface. deet is a replacement for this fourth component. Using deet does not directly involve changes to any of the other components, but implementing deet did induce additions to the nub and to the symbol-table format beyond their original designs.

deet is written in tksh, which includes a C library that can be used to manipulate the state of the Tcl interpreter, such as reading and writing variables and creating new built-in commands. tksh can run any library written on top of the Tcl library, which includes the Tk graphics library. Thus, Tk commands, like button and pack, can be invoked from tksh scripts.

tksh should be thought of as an extension to Tcl rather than as an alternative to it. tksh allows Tcl scripts to be run directly with the source command. Tcl scripts share variables and functions with tksh, allowing Tcl scripts to work with shell scripts.

tksh is used as the debugging language for deet primarily because of its strengths as an interactive command language. Debuggers are interactive programs. deet takes advantage of the interactive facilities of tksh, such as command-line editing, job control and pipelines. Using the command-line interface to deet feels like using a shell because the debugger itself is an extension of ksh. tksh also offers two familiar, high-level languages. Many programmers already know how to write shell and Tcl scripts, which is 90% of what's needed to use deet. However, a perl or python programmer could rewrite the deet front end and still use existing nub implementations.

deet also includes additional built-in commands for debugging. deet's code is simpler to understand than the corresponding C code would be, because it's written in a high-level language. deet can also can be modified during a debugging session to suit specific applications.

3.2 The Nub Interface

The nub interface is designed to be as small as possible while supporting the fundamental debugging operations common to all debuggers [6]. Figure 5 summarizes the complete API. The nub does not support high-level facilities, such as expression evaluation or specific symbol-table formats, because these facilities can be implemented by other interfaces or by debuggers themselves.

_Nub_set and _Nub_remove set and remove breakpoints, which are specified by a file name, line number, and character position. Unlike most debuggers, breakpoints specify the locations of expressions, not lines. So, for example, it is possible to set a breakpoint on the increment part of a C for loop.

_Nub_src accepts incomplete breakpoints, in which any of the file name, line number, or character position are omitted, and invokes a debugger callback on all possible breakpoints that "match" the incomplete one. deet

`_Nub_init`	initialize the nub
`_Nub_set`	set a breakpoint
`_Nub_remove`	remove a breakpoint
`_Nub_src`	visit breakpoints with a given pattern
`_Nub_frame`	move to a specific stack frame
`_Nub_fetch`	read the target's memory
`_Nub_store`	write the target's memory

Figure 5: The nub interface

uses this function to determine which breakpoint to set when a user clicks on a line.

`_Nub_fetch` and `_Nub_store` access the target's memory. They accept a buffer address, a byte count, and an address space identifier, and read/write data from/to the target. The address space identifier may specify an operating-system address space, such as the text or code segments. It can also specify logical address spaces that may not be part of the target, like the symbol table, for example. It is the nub's responsibility to access the appropriate data. Debuggers can view all data about the target as if they were stored in memory.

`lcc` emits machine-independent symbol tables in the target's address space, and `_Nub_fetch` reads these data. Another `cdb`-specific, but machine-independent, interface provides a higher-level view of the C symbol table as an inverted tree of symbol objects. If we were using the nub with, say, Modula-3, this high-level interface would have to be replaced with one specific to Modula-3, and that interface would use `_Nub_fetch` to read the symbol table generated by the Modula-3 compiler. There's nothing special about `lcc`; other C compilers could be used given an appropriate nub implementation. It's the nub that's the critical component, not the compiler.

3.3 The `deet` Nub Interface

`deet` includes versions of the nub and symbol-table functions for use with Tcl or `tksh`. These `tksh` commands differ from the C routines in two ways: They are at a higher level, because they manipulate source-level symbols, types, and values, and they accept and return strings, so that they can be used in Tcl or `tksh` scripts. The complete list appears in Figure 6.

`deet_breakpoint` is a combination of `_Nub_set`, `_Nub_remove`, and `_Nub_src`. The `-set` option sets all of the given breakpoints, which might be incomplete; that is, *file* can be `""`, and *line* and *character* can be zero. The `-delete` option removes breakpoints, and the `-list` option lists possible breakpoints.

`deet_frame` is equivalent to `_Nub_frame`: With no arguments, it returns the current frame as a Tcl list containing the frame number, the function name, and a file, line number, character number triple that gives location of execution within that frame. With an integer argument n, `deet_frame` makes frame n the current frame and returns the null string. Frames are numbered from the top of the stack, beginning with zero.

`deet_getval` and `deet_putval` commands are similar to `_Nub_fetch` and `_Nub_store`, but require type information to be specified along with the value, because Tcl deals only with strings. Tcl cannot, for example, deal directly with binary floating-point values or with structures. Types are specified by type identifiers, which are just generated strings. `deet_getval` returns a string representation for the value of *type* at *address*, and `deet_putval` writes the *value* of *type* to locations beginning at *address*.

`deet_sym` and `deet_type` return symbol table data. A symbol-table entry is a Tcl list { *name*, *type*, *address* }. `deet_sym`'s `-all` option returns a list of all of the symbols in the target; that is, a list of three-element lists. The `-files` option returns a list of all of the source files in the target. The `-locals` and `-params` options return lists of the locals and parameters for the current frame. The `-name` *name* returns the symbol-table entry for *name*, or an error if *name* is not a visible symbol.

`deet_type` returns a string describing the type represented by the identifier *type*. If *type* represents int, `deet_type` returns `"int"`. Similarly, if *type* represents T `*`, E `[`n`]`, or a structure type, `deet_type` returns, respectively, the type identifier for T, n and the type identifier for E, and a list of names and type identifiers for the fields.

NeD [10] is another debugger built on a set of debugging primitives. This set is larger than the set of nub functions and the NeD primitives are at a somewhat higher level. NeD's primitives are written in Tcl extended with a set of debugging functions. While these functions present a nearly platform-independent interface, their implementation appears to be platform-dependent and perhaps nontrivial. Also, NeD has no user interface per se; it uses Tcl in the same way as `deet`

`deet_open`	initialize the target
`deet_breakpoint { -set | -delete | -list }` *file line character*	set, remove, and list breakpoints
`deet_frame` [n]	get/set current frame
`deet_getval` *type address*	read a value of *type* from *address*
`deet_putval` *type address value*	write the *value* of *type* to *address*
`deet_continue`	resume execution
`deet_sym { -all | -files | -locals | -params | -name` *name* `}`	finds the symbol-table entries
`deet_type` *type*	get *symbol*'s type information

Figure 6: `deet`'s nub interface

uses the nub functions, while `deet` uses Tcl as its user-interface language, as illustrated in the next section.

4 Programming in `deet`

Much of `deet` itself is written in Tcl and `tksh`, using the `deet_*` nub commands described above. Users can extend `deet` by writing Tcl and `tksh` commands; for example, features like conditional breakpoints and non-trivial program queries can be written in `tksh`. `deet` can also be extended by external programs. This section illustrates some typical extensions.

Simple extensions can be written directly in `tksh`. For example, the following script displays all of the null elements in an array, the name of which is supplied as an argument.

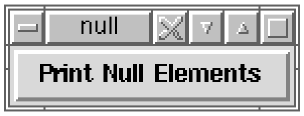
```
function nullElements {
  typeset arr=$1
  integer i s=$(arraySize $arr)
  for (( i=0 ; i < s ; i++ )); do
    if [[ $(var "$arr[$i]") == 0x0 ]]
    then
      print "Element $arr[$i] null"
    fi
  done
}
```

`nullElements` uses two external `tksh` functions: `arraySize`, which returns the number of elements in an array, and `var`, which returns the value of a variable. These functions are provided as part of `deet`. The for loop visits each element of the array specified by the first argument, retrieves its value, and prints the array name and index of the null elements.

User-defined functions can also manipulate `deet`'s interface. For example, if we're checking repeatedly for null elements in `hashtable`, we can construct a button to do the job in one click:

```
toplevel .null
pack $(button .null.b \
  -text "Print Null Elements" \
  -command "nullElements hashtable")
```

This code builds the button:



Tcl scripts can be invoked with the `source` command, which is a `tksh` built-in. `source` uses the Tcl parser to parse its input, and uses `tksh` variables and functions in variable and command substitutions. Here's a simple example:

```
function foo {
  X=37
  print "$(bar test)"
}

source <<'EOT'
proc bar {args} {
  global X
  set X [expr $X + 1]
  return "bar: args: $args, X: $X"
}
EOT
```

A call to `foo` prints

```
bar: args: test, X: 38
```

Note that the Tcl procedure `bar` can use and modify the shell variable `X`. Tcl source code can also invoke `tksh` functions and built-ins.

`deet`'s name space is separate from the target's name space. Accessing a target variable from a `tksh` script requires a special function, `var`, which uses the target's symbol table to lookup the variable name and retrieve its value. var is sufficient for one-shot lookups, but it's tedious for repeated uses of specific target variables. For these uses, deet provides linkvar *name*, which creates a new shell variable that is essentially an alias for the target variable *name*. linkvar is implemented with *discipline functions*, which are similar to trapped variables in SNOBOL4 [5]. A discipline function is a shell function that is associated with a variable, and that function is invoked whenever the variable is read or written. Thus, associating the function

```
function foo.get {
  foo="$(var foo)"
}
```

with foo arranges for the target variable to be fetched every time the shell variable foo is read.

deet's capabilities are easily extended by writing Tcl and tksh scripts that use the built-in debugger commands. An important advantage using a shell as the debugging language is that the shell can use *any* external tool. For example, it's relatively easy to extend deet to display linked data structures graphically as directed graphs. This feature is similar to that provided by the Data Display Debugger (ddd) [16], but the implementation is much simpler, because deet uses existing tools instead of building its own facilities. deet runs dotty, a program for drawing directed graphs [9], to draw the graph, sending it the appropriate input for the data structure of interest. Figure 7 shows an example of dotty's output. The tksh script that invokes dotty is only about 60 lines of code, and it handles any linked data structure.

5 Implementation

deet is written in tksh; each deet command is implemented as one or more tksh functions that call the built-in Tcl nub commands. An example is the b function shown in Figure 8, which uses deet_breakpoint to set breakpoints. The size of this function is as important as its details: most debugging features are easily implemented in tens of lines of tksh code.

b begins by converting its first argument into file, line number, and character number values. When an incomplete breakpoint is specified, some of these values will be converted to null values. For example, b 8 causes the cvtbp function for the argument 8 to become the value of line and for file and char to be null. Next, b invokes deet_breakpoint -list to list all breakpoints matching the incomplete breakpoint. If there is more than one match, a list of possible breakpoints is displayed and no breakpoints are set. If there are no matches, a diagnostic is issued. Finally, if there is exactly one match, that breakpoint is set. The associative array breakpoint keeps track of the set breakpoints. The nub doesn't keep track of breakpoints because it is designed to do as little as possible. If the second argument specifies a condition for the breakpoint, it's stored as the value for the breakpoint array entry. Finally, the source window (if it exists) is updated to highlight the set breakpoint.

The nub interface can set and remove breakpoints, but it cannot single-step the target [6]. deet's step function implements single-stepping by setting and removing breakpoints:

```
function step {
  if [[ $cdbMode != "step" ]]; then
    deet_breakpoint -set "" 0 0
  fi
  cdbMode=step
  cdbgo         # resume execution
}
```

Calling deet_breakpoint with null values for the file, line number, and character number sets every breakpoint. Implemented naively, setting every breakpoint is expensive in large programs. But the nub could recognize this special case and use a more efficient implementation. As described in Section 6.2, our implementation of the Tcl nub functions on top of gdb exploits this possibility.

6 Replacing the Nub

An important aspect of deet's "piece-parts" design is that superior replacements could be used for each part without disturbing the others. For example, a more efficient, machine-specific nub could be used in place of cdb's machine-independent nub; or a better or more familiar user interface could be used.

To demonstrate this flexibility, we've implemented three alternative versions of deet's pieces: a version of the nub for Java [1], a nub that works by communicating with gdb, and a replacement for the user-interface component that emulates gdb's command-line interface. These limited experiments also reveal strengths and weaknesses in the nub-based design. If gdb cannot emulate the nub, for example, then a simple nub offers facilities beyond those of some popular debuggers. If the nub cannot support gdb, then the nub is missing some important facilities.

6.1 A Nub for Java

The Java Developer's Kit contains a debugging package (a set of classes) that can be used to explore and con-

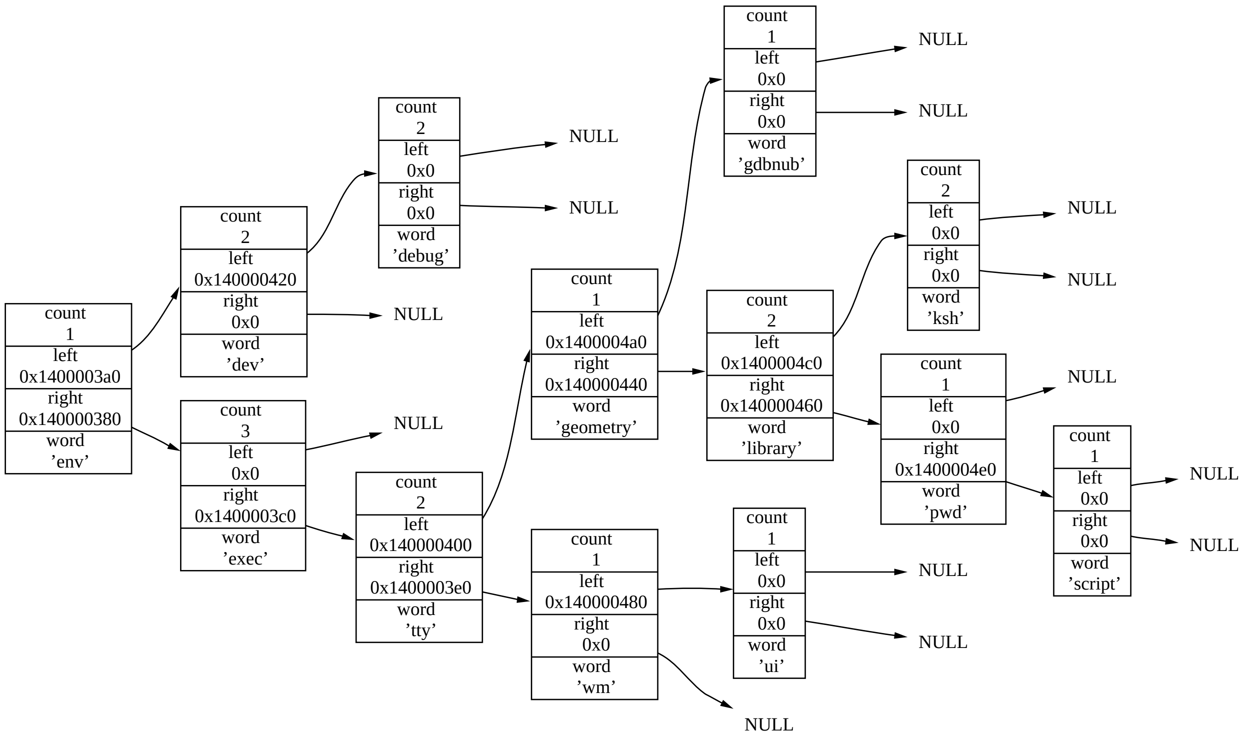


Figure 7: Tree generated with `deet` and `dotty`

```
function b {  # breakpoint [action]
  integer char=0 line=0
  typeset file point="$1" action="${2-':'}" msg
  eval set -- $(cvtbp "$point")
  file="$1" line="$2" char="$3"
  typeset bp="$(deet_breakpoint -list "$file" "$line" "$char")"
  eval set -- $bp
  if (( $# > 1 )); then
    msg="Pick one of $bp"
  elif (( $# < 1 )); then
    msg="No breakpoint in on line $line char $char"
  else
    deet_breakpoint -set "$file" "$line" "$char"
    set -- $1
    breakpoint["$1:$2.$3"]="$action"
    [[ $CdbWindow ]] && TextDispBpOn $2
  fi
  [[ $msg ]] && print -- "$msg"
}
```

Figure 8: Implementation of the `b` command

trol the state of a target Java program. These classes are designed to support a variety of Java debuggers. Java comes with jdb, a simple command-line debugger, that is implemented with the debugging package, and this package is intended to be used to write more sophisticated graphical debuggers. The package works by spawning an instance of the Java runtime with the target and communicating with it via message passing.

Although the Java nub itself could spawn the runtime and the target, it's simpler to use the debugging package. Implementing a nub for Java required writing the nub interface in terms of Java's debugging methods, which takes only a couple hundred lines of Java. The routines in the nub read messages from a socket, process them using the methods in the Java debugger package, and write the result messages back to the socket. A central method reads messages, decodes them, and calls the appropriate methods for each message.

Figure 9 shows the method frameCmd, which performs the same task as deet_frame. When called with an argument, frameCmd sets the current frame to the number specified by the argument by finding the current frame with the getCurrentFrameIndex method and calling the up and down methods as needed. If there is no argument, frameCmd returns information about the current frame. This information is returned by calling methods in the debugging classes RemoteThread, RemoteStackFrame, and RemoteClass.

Thus, with a couple hundred lines of Java code, deet can be used to debug Java programs with the same set of features that are used to debug C. Unfortunately, the nub interface does not currently support threads, which limits the usefulness of the Java debugger.

A similar approach can probably be used on any system that has a debugging interface. For example, deet can be ported to Windows by implementing a nub in terms of Microsoft's debugging API.

6.2 Using gdb as a Nub

We've used gdb to build a variant of deet that uses gdb as the nub, and a variant that uses gdb as the user interface.

Figure 10 shows how gdb replaces the nub. gdbnub communicates with gdb, which runs as a separate process. gdbnub translates nub function calls into gdb commands, sends these commands to gdb, and parses gdb's responses. Another approach would have been to modify gdb's code, but past experience shows that modifying gdb is a painstaking process [4]. Using tksh makes the approach illustrated in Figure 10 *much* simpler: the implementation takes only about 500 lines.

The only nub feature that was not possible to implement with gdb was setting breakpoints on any expres-

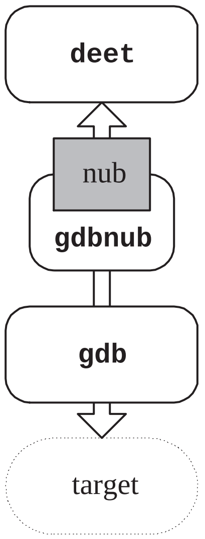


Figure 10: Emulating the nub with gdb

sion. Some nub functions were relatively easy to implement, but not efficiently. For example, gdb doesn't provide a way to list all of the possible breakpoints in a file. This was implemented by attempting to set a breakpoint at every line in each file, and checking which breakpoints were successfully set. Fortunately, listing all of the possible breakpoints is rarely done; listing the breakpoints in a specific line is the common usage.

As described at the end of Section 5, deet steps through a program by issuing the nub command to set every breakpoint in the target, which is very inefficient. gdbnub takes advantage of gdb's single-stepping feature by using the deet_breakpoint implementation shown in Figure 11. When null values are passed to the -set option, a variable is set that puts the gdbnub into "stepping" mode when execution is resumed. Similarly, when null values are supplied with the -delete option, gdbnub reverts to "continuation" mode, and all temporary breakpoints are removed.

When gdb is used as the nub, deet can be thought of as a graphical front end to gdb. It provides facilities similar to ddd's, which is also a front end for gdb. However, ddd is as not programmable.

6.3 A Nub for gdb

Implementing gdb on top of the nub is difficult, because gdb is a huge program and has a large number of features. Some of the features in gdb are inherently absent in the nub. For instance, gdb allows the target to be examined at the machine level; gdb can examine registers and single-step instructions. The nub interface is machine-independent, so it cannot provide these

```
public void frameCmd(RemoteThread t, String args[]) throws Exception {
  if (args.length == 2) {
    int oldFrame = t.getCurrentFrameIndex();
    int newFrame = Integer.parseInt(args[1]);
    try {
      if (oldFrame < newFrame)
        t.up(newFrame-oldFrame);
      else if (oldFrame > newFrame)
        t.down(oldFrame-newFrame);
    } catch (ArrayIndexOutOfBoundsException e) {
      outputError();
    }
  } else {
    RemoteStackFrame s = t.getCurrentFrame();
    RemoteClass c = s.getRemoteClass();
    outputItem(t.getCurrentFrameIndex());
    outputItem(c.getName() + "." + s.getMethodName());
    outputItem(c.getSourceFileName());
    outputItem(s.getLineNumber());
    outputItem("0");  /* No support for char position */
  }
}
```

Figure 9: Implementation of `deet_frame` for Java

```
function deet_breakpoint {
  typeset i action=list
  case $1 in
  -l*)  shift ;;
  -s*)  action=set ; shift
    if [[ $1 = "" && $2 = "0" && $3 = "0" ]]; then
      sendCommand "b main"
      CONT_CMD="step"
      return 0
    fi ;;
  -d*)  action=delete ; shift
    if [[ $1 = "" && $2 = "0" && $3 = "0" ]]; then
      sendCommand "delete"
      for i in "${!GdbBreakpoint[@]}"
      do unset GdbBreakpoint[$i]
      done
      CONT_CMD="cont"
      return 0
    fi ;;
  esac
  ...
}
```

Figure 11: Implementing single-stepping

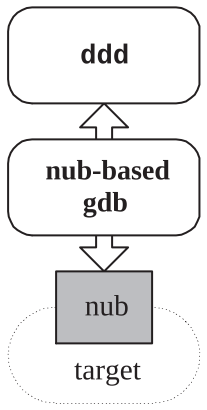


Figure 12: Running `gdb` with the nub

machine-dependent features. Similar caveats apply to data watchpoints, which `gdb` supports on machines with the appropriate hardware. `gdb` also supports features that are irrelevant to debugging, per se, such as controlling terminal modes and displaying `gdb`-specific online help.

Our third experiment thus focuses on only those `gdb` features used by a graphical front end, like `ddd`. Figure 12 shows the organization of this experiment.

The implementation of `gdb` using the nub was written with `deet` commands. `gdb`'s `frame` command illustrates the general approach. `frame` controls the current frame in the stack of the target. Invoking `frame` with an argument directs `gdb`'s attention to a specific frame. If no argument is specified, the current function with its arguments and location is displayed. For example:

```
(gdb) frame
#0  lookup (word=0x11ffff8e0 "a",
  p=0x140000010) at test/lookup.c:15
```

Figure 13 shows the `tksh` implementation of `frame`. The function uses `deet_frame` to move the nub's attention to a new frame. It also uses `deet_sym` and `deet_getval` to fetch and display the frame's parameters and their values.

This implementation of `gdb`, although incomplete, is only around 1,000 lines of `ksh`. It implements enough features to support `ddd`. `gdb` features that were not implemented include:

- Debugging a target that is already running, which `gdb` can on machines where this is possible.
- Invoking target functions from the debugger; the nub doesn't support this feature, because a separate evaluation facility can support it [13].
- Examining core dumps; this feature could be supported by writing a nub specifically for browsing core dumps.
- Interrupting a running target.
- Handling signals.

7 Discussion

`deet`'s front end runs on any machine on which `tksh` runs, which currently includes virtually all UNIX variants, Windows NT and Windows 95. Graphical debuggers that work consistently under both UNIX and Windows are scarce, and having a uniform interface can be important. Programmers writing code for multiple platforms can debug applications without having to learn multiple environments. Another advantage of a uniform interface is that one set of debugging scripts is often sufficient for all platforms.

The nub hides of most of the difficult portability issues. `deet` is available on all of `lcc`'s platforms, because its nub interface is machine-independent. `deet` is also available on platforms that support `gdb`, because it can use the nub that runs on top of `gdb`. `deet` can be made available on other platforms by writing a new, platform-specific nub. Typical nub implementations take less than a thousand lines of code, so they aren't trivial, but the effort required is tiny compared to porting a machine-specific debugger.

`deet` demonstrates that it is possible to build a usable debugger with a graphical user interface from simpler components, and, as the `dotty` example illustrates, that the result is more than just the sum of the parts. `deet` also confirms `cdb`'s premise that most of a debugger is machine-independent, and that the fundamental machine-specific debugging facilities can be encapsulated in a small, machine-independent nub interface.

`deet` doesn't have all of the features offered by PC debuggers and UNIX debuggers like `gdb`. But it does provide the most important ones—at a fraction of the implementation cost. `deet` is about 1,500 lines of `tksh` code, and the machine-independent nub and related compiler support (in `lcc`) total around 800 lines of C. These 2,300 lines of code are orders of magnitude smaller than `gdb`'s 150,000 lines and `ddd`'s 90,000 lines.

Programmers interact with most debuggers in the target's source language plus a few debugger-specific commands. For example, programmers throw C expressions at `gdb` to browse the state of a buggy target. The advantage of this approach is that programmers don't have to learn another language to use the debugger. But, as Acid [15] and Duel [4] demonstrate, exploring a program's state is fundamentally different than writing the

```
function frame {  # [num]
  [[ $1 != "" ]] && deet_frame $1 2> /dev/null
  set -- $(deet_frame)
  typeset num=$1 name=$2 file=$3 line=$4 char=$5
  typeset params="$(deet_sym -params)" p result
  result="#$num  $name("
  eval set -A parm $params
  for p in "${parm[@]}"; do
    set -- $p
    result="$result${1##*:}=$(deet_getval "$2" "$3"), "
  done
  print -- "${result%', '}) at $file:$line"
}
```

Figure 13: `ksh` implementation of `gdb`'s `frame` command

program in the first place, and this exploration can be done much more effectively in a higher-level language. `deet` also supports this view; Tcl and `tksh` seem to be better languages for writing debugging code than languages like C and C++. Similar comments may apply to other high-level scripting languages, like Perl.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.

[2] M. Bolsky and D. Korn. *The New KornShell Command and Programming Language.* Prentice Hall, Upper Saddle River, NJ, second edition, 1995.

[3] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, Menlo Park, CA, 1995.

[4] M. Golan and D. R. Hanson. DUEL—a very high-level debugging language. In *Proceedings of the Winter USENIX Technical Conference*, pages 107–117, San Diego, CA, Jan. 1993.

[5] D. R. Hanson. Variable associations in SNOBOL4. *Software—Practice and Experience*, 6(2):245–254, Apr. 1976.

[6] D. R. Hanson and M. Raghavachari. A machine-independent debugger. *Software—Practice and Experience*, 26(11):1277–1299, Nov. 1996.

[7] D. G. Korn. ksh: An extensible high level language. In *Proceedings of the Very High Level Languages Symposium (VHLL)*, pages 129–146, Santa Fe, NM, October 1994.

[8] J. L. Korn. Tksh: A Tcl library for KornShell. In *Proceedings of the USENIX Tcl/Tk Workshop*, pages 149–159, Monterey, CA, July 1996.

[9] E. Koutsofios and S. C. North. Applications of graph visualization. In *Proceedings of Graphics Interface 1994 Conference*, pages 235–245, Banff, Canada, May 1994.

[10] P. Maybee. NeD: The network extensible debugger. In *Proceedings of the Winter USENIX Technical Conference*, pages 145–153, San Antonio, TX, July 1992.

[11] Microsoft Corp., Redmond, WA. *Microsoft Visual C++, Reference Volume II*, 1993.

[12] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.

[13] N. Ramsey and D. R. Hanson. A retargetable debugger. *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 27(7):22–31, July 1992.

[14] R. M. Stallman and R. H. Pesch. Using GDB: A guide to the GNU source-level debugger, GDB version 4.0. Technical report, Free Software Foundation, Cambridge, MA, July 1991.

[15] P. Winterbottom. Acid: A debugger built from a language. In *Proceedings of the Winter USENIX Technical Conference*, pages 211–222, San Francisco, CA, Jan. 1994.

[16] A. Zeller and D. Lütkehaus. DDD — a free graphical front-end for UNIX debuggers. *SIGPLAN Notices*, 31(1):22–27, January 1996.