A Debugger for Tcl Applications

Don Libes
National Institute of Standards and Technology

Abstract

Tcl is a language specifically intended for generic application control. By using it, application programmers escape the dilemma of whether to design sophisticated application-specific languages or whether to build tools more quickly but that are limited in flexibility. Tcl is easy for application programmers to use, however, up to now, there has been no general-purpose debugger for application users.

This paper describes an implementation of a debugger for Tcl applications. The debugger has a typical front-end but with some extremely unusual commands, in part because of the features and limitations of Tcl. The debugger is modeless, allowing users to issue Tcl and application commands along with debugger commands. Each type of command may invoke the other, allowing debugging to be programmed, dynamically or in advance.

The debugger is written in C and is very fast. When linked in but not used, it does not slow applications at all. The debugger requires no modifications to the Tcl core, and can be plugged into applications with little effort.

Keywords: Tcl, Tk, Tool Command Language, Expect, debugger, interpreter

Introduction

Tcl [1][2] is a language specifically intended for generic application control. In some applications it dominates the user's view of the application. Expect [3][4] is a good example of this. Expect is always executing Tcl statements – usually from a script. When the script ends, Expect ends.

This type of application is common. With this heavy reliance on Tcl, it is useful to have a debugger that understands Tcl. For example, it should be able to single-step and print values at user discretion.

The debugger described herein works in exactly this way. It works best with applications like Expect that sequentially execute Tcl commands, although it can also be used with any Tcl application.

This paper has three parts. The first part describes the debugger in terms of how it is used in a typical application. The second part of the paper describes how the debugger can be integrated into Tcl applications. The third part describes the design and implementation of the debugger.

View by the Application User

This section of the paper is a debugger tutorial which shows the application user's view of the debugger. For the sake of concreteness, Expect [1] will be used as the application. However, any other application using the debugger will work similarly.

Starting the Debugger

The debugger is initially invoked in an application-dependent way. In Expect, the debugger is started by using the flag "-D 1". For example:

```
% expect -D 1 script
```

If the system supports the #! mechanism, the script may also be started as:

```
% script -D 1
```

In either case, additional arguments may be supplied as usual.

The user is prompted for a command. At this point either Expect commands, Tcl commands, or debugger commands may be entered. This is true at all other times that the user is prompted as well. The debugger is modeless.

The following Tcl commands illustrate that the debugger evaluates Tcl commands as usual.

```
expect2.1> set m {a b c}
a b c
expect2.2> llength $m
3
expect2.3>
```

The command prompt is changeable by the application or user. Here, the second number is the Tcl history identifier. The first number is the depth of the evaluation stack.

In the context of a script, the initial depth of the evaluation stack is 1 but the debugger always introduces a new level to the stack. Hence, we see a "2" in the prompt.

Expect also allows the application to take initial control. By using the flag "-D 0", the application runs until the user presses ^C at which time the debugger begins running. The remainder of this tutorial will assume that the debugger has started up immediately from the flag "-D 1".

Command Overview and Philosophy

The debugger commands are:

Name	Description
s	step into procedure
n	step over procedure
r	return from procedure
b	set, clear, or show breakpoint
c	continue
w	show stack
u	move scope up
d	move scope down
h	help

The debugger commands are all one letter. Short procedure names are desirable in an interactive-only application such as the debugger. In contrast, scripted applications rarely use one-letter commands. The chances of name conflict between the debugger and scripted applications are very low.

The command names are very similar and, in some cases, identical to other popular debuggers (gdb, dbx, etc.). Existing Tcl procedures are directly usable so there are no new commands, for example, to print variables since Tcl already provides such commands (e.g., set, puts, parray). The intent of the debugger is that it should be easy to learn and use, and otherwise stay out of the way.

The debugger uses the application's top-level interactor. In the case of Expect, for example, the debugger uses Expect's interact command which prompts for commands and evaluates them.

For the purposes of describing the debugger commands, the following script is assumed to be named *debug-test.exp*.[1] The script doesn't do anything useful. It merely serves to illustrate how the debugger is used.

```
set b 1

proc p4 {x} {
      return [
           expr 5+[expr 1+$x]]
}
```

1. Italics indicate something being defined.

```
set z [
   expr 1+[expr 2+[p4 $b]]
]

proc p3 {} {
      set m 0
}

proc p2 {} {
      set c 4
      p3
      set d 5
}

proc p1 {} {
      set a 2
      p2
      set a 3
      set a 5
}


p1
set k 7
p1
```

If the debugger is started at the beginning of the script, no statements have been executed. Tcl and application commands have global scope.

```
% expect -D 1 debug-test.exp
1: set b 1
expect2.1>
```

When a new command is about to be executed, the debugger prints the command. It is preceded by the evaluation stack level. “set b 1” is the first line in the script. It has not yet been executed. “info exists” confirms this.

```
expect2.1> info exists b
0
```

The “n” command – “Next”

The *n* command executes the pending command – in this case “set b 1” – and displays the next command to be executed.

```
expect2.2> n
1: proc p4 {} {
        return [
            expr 5+[expr 1+$x]]
}
expect2.3> info exists b
1
```

The command “info exists b” confirms that b has been set. The procedure p4 is about to be defined.

```
expect2.4> n
4: p4 $b
expect5.5>
```

The procedure p4 has now been defined. The next command to be executed is p4 itself. It appears in the statement:

```
set z [
      expr 1+[expr 2+[p4 $b]]
]
```

The three sets of braces introduce three new levels on the evaluation stack, hence the stack level in which p4 is about to be executed is shown as “4”.[1]

Notice that the evaluation stack level does not affect the scope. We are still in the top-level scope and b is still visible.

```
expect5.5> info exists b
1
```

The argument to p4 is $b. The value of this can be evaluated by using set or puts.

```
expect5.6> set b
1
expect5.7> puts $b
1
```

Another n command executes p4, popping the stack one level. Additional n commands continue evaluation of the “set z” command, each time popping the stack one level.

```
expect5.8> n
3: expr 2+[p4 $b]
expect4.9> n
2: expr 1+[expr 2+[p4 $b]]
expect3.10> n
```

1. Whether the word “stack” refers to procedure call stack or evaluation stack should either be explicit or clearly implied by context.

```
1: set z [
      expr 1+[expr 2+[p4 $b]]
]
expect2.11>
```

The "s" command – "Step"

The n command executes a procedure atomically. It is possible to step into a procedure with the *s* command.

We'll rewind this scenario to just before p4 is about to be executed.[1]

```
4: p4 $b
expect5.5> s
7: expr 1+$x
expect8.6>
```

"expr 1+$x" is the first command to be executed inside of p4. It is nested inside of two brackets, plus the procedure call of p4, so the stack level is increased by three.

After the s command, the debugger stops before the first command in the procedure and waits for more interactive commands.

If the command that is about to be executed is not a procedure, then s and n behave identically.

Both s and n take an optional argument in the form of a number describing how many commands to execute.

For example:

```
s 2
s 100
s $b
s [expr 2+[p4 $b]]
```

The arguments are evaluated according to the usual Tcl rules because s and n are commands known to Tcl.

The debugger will not interrupt procedures invoked from the command-line. This is usually the desired behavior, although it is possible to change this.

1. There is no actual command to "rewind" commands, alas.

The "w" Command – "Where"

In the current scenario, we are about to execute "expr 1+$x" in the procedure p4. We can remind ourselves of this by displaying the stack of procedure scopes using the *w* command.

```
7: expr 1+$x
expect8.6> w
 0: expect -D 1 debug-test.exp
*1: p4 1
 7: expr 1+1
```

The first line describes scope 0. This is the top-level scope of the file itself, and the command used to invoke the program is shown. The second line describes scope 1 which is the invocation of procedure p4. The last line is not a scope but just repeats the evaluation stack level and the command about to be executed.

Notice that when w prints commands, they are displayed using the literal values of each parameter. In contrast, when the debugger automatically prints out the next command to be executed, the command is printed as it was originally entered in the script. For example, the debugger initially stopped and printed "expr 1+$w", but the same instruction shows as "expr 1+1" in the output from the w command.

The Current Scope

Executing fourteen steps (via "s 14") brings us to the first command in procedure p3.

```
expect8.8> s 14
4: set m 0
expect5.9> w
 0: expect -D 1 debug-test.exp
 1: p1
 2: p2
*3: p3
 4: set m 0
```

The asterisk denotes that p3 is the *current scope*. We can now execute Tcl commands appropriate to the scope of p3. This includes commands such as global, uplevel, and upvar.

```
expect5.10> uplevel {set c}
4
```

The "u" and "d" commands – "Up" and "Down"

The current scope can be changed by the u and d commands. *u* moves the current scope up, while *d* moves it down. Interactive variable accesses always refer to the current scope.

```
expect5.11> u
expect5.12> w
 0: expect -D 1 debug-test.exp
 1: p1
*2: p2
 3: p3
 4: set m 0
expect5.13> set c
4
```

Both u and d accept an argument representing the number of scopes by which to move. For example, "u 2" moves from scope 2 to scope 0.

```
expect5.14> u 2
expect5.15> w
*0: expect -D 1 debug-test.exp
 1: p1
 2: p2
 3: p3
 4: set m 0
```

An absolute scope is also accepted in the form of "#" followed by a scope number, such as "#3".

```
expect5.16> u #3
expect5.17> w
 0: expect -D 1 debug-test.exp
 1: p1
 2: p2
*3: p3
 4: set m 0
```

When an absolute scope is named, either u or d may be used, irrespective of which direction the new scope lies.

Moving the scope does not affect the command that is about to be executed. If a command such as s or n is given, the current scope is automatically reset to whereever is appropriate for execution of the new command.

The "r" Command – "Return"

The *r* command completes execution of the current procedure. In other words, it stops after the current procedure returns.

```
expect5.18> r
3: set d 5
expect4.19> w
 0: expect -D 1 debug-test.exp
 1: p1
*2: p2
 3: set d 5
expect4.20> r
2: set a 3
expect3.21> w
 0: expect -D 1 debug-test.exp
*1: p1
 2: set a 3
expect3.22> r
1: set k 7
expect2.23> w
*0: expect -D 1 debug-test.exp
 1: set k 7
expect2.24> r
nowhere to return to
```

The "c" Command – "Continue"

The *c* command lets execution of commands continue without having to single-step. In the scenario so far, given a command anywhere, the program would continue until the script ends and the shell prompt appears.

```
expect2.25> c
%
```

The c command is also useful in other ways. After setting breakpoints, the program can be continued until it hits a breakpoint. The program can also be continued until a signal occurs, such as by the user pressing ^C.

The "b" Command – "Breakpoint"

Prior commands have shown how to execute a fixed number of commands or procedure calls. In contrast, breakpoints provide a way to stop execution upon a condition. The conditions include:

- line number and filename matching
- expression testing
- command and argument name matching

Breakpoint by Line Number and Filename[1]

Line numbers and filenames are the most common way to specify a breakpoint. This form is correspondingly the most compact. For example the following command causes execution to break before executing line 7.

```
expect2.26> b 7
0
```

After creation of a breakpoint, an integer identifying the breakpoint is printed. The reason for this will be described later.

By default, the line number refers to the file associated with the current scope. A filename may be used to refer to a different file. A colon is used to separate the filename and line number.

```
expect2.27> b foo.exp:7
```

Breakpoint by Expression

It is possible to break at a line only when an expression is true. For example, the following command causes execution to break at line 7 only when foo is greater than three.

```
expect2.28> b 7 if {$foo>3}
```

Expressions are the usual Tcl syntax and may be arbitrarily complex.

No breakpointing occurs inside of the evaluation of breakpoint expressions (unless another breakpoint dictates this).

Breakpoint by Pattern Match

It is also possible to define breakpoints by pattern matching on the command or arguments. Regular expressions are introduced by the flag "-regexp" (commonly abbreviated "-re")[2]. The following command stops if the string p4 appears within the command:

```
expect2.29> b -re "p4"
0
```

With our sample file, we can see the results of this:

```
% expect -D 1 debug-test.exp
1: set b 1
expect2.1> b -re "p4"
0
expect2.2> c
breakpoint 0: -re "p4"
1: proc p4 {x} {
        return [
           expr 5+[expr 1+$x]]
}
expect2.3> c
breakpoint 0: -re "p4"
4: p4 $b
expect5.4> c
breakpoint 0: -re "p4"
3: expr 2+[p4 $b]
expect4.5> c
breakpoint 0: -re "p4"
2: expr 1+[expr 2+[p4 $b]]
```

The first breakpoint occurred upon the definition of p4. The second occurred when p4 was called. Two more breakpoints occurred only because p4 was mentioned in the command.

With appropriate regular expressions, any one of these can be selected by itself. For example, to stop only on definitions:

```
expect2.1> b -re "proc p4 "
```

To stop only on a call to p4 itself:

```
expect2.2> b -re "^p4 "
```

To stop only on commands which call p4:

```
expect2.3> b -re "\\\[p4 "
```

The complexity of this last example is, perhaps, somewhat ameliorated by the unlikelihood of it ever being used. It is more shown simply for completeness. The point is, the ability to match on regular expressions is extremely powerful.

1. Breakpoints by line number and filename are not currently supported. See "Line Numbers" on page 13 and "Current Limitations and Future Work" on page 15.

2. The debugger permits all flags to be abbreviated to the smallest unique prefix. For example, "-regexp" can actually be abbreviated "-r". The usual quoting conventions around patterns should be observed. In this example, the quotes around p4 can be omitted.

Multi-line patterns may be matched in the usual way – using characters such as \n and \r.[1]

Glob-style matching is available by using the flag -glob instead of -regexp. It works exactly as in Tcl's case command. Since glob matches an entire string by default, the equivalents to the previous example look slightly different. Note the asterisks.

To stop only on definitions:

```
expect2.4> b -glob "proc p4 *"
```

On calls to p4:

```
expect2.5> b -glob "p4"
```

On commands which call p4:

```
expect2.6> b -glob "*\\\[p4 *"
```

Expressions can be combined with patterns just as if they were with line numbers. For example, to break on a call to p4 only when foo is greater than three:

```
expect2.7> b -glob p4 if {$foo>3}
```

Regular expression patterns save the strings which matched any patterns in the array *dbg*. The part of the command matched by the entire pattern is saved in $dbg(0). Up to 9 parenthesized subpattern matches are stored in $dbg(1) through $dbg(9).

For example, the name of a variable being set can be accessed as $dbg(1) after the following breakpoint:

```
expect2.8> b -re {^set ([^ ])+ }
```

This can be used to construct more sophisticated breakpoints. For example, the following breakpoint occurs only when the variable being set was already set.

```
expect2.9> b -re {^set ([^ ])+ }
if {info exists $dbg(1)}
```

Breakpoint Actions

Breakpoints may trigger actions. The default action prints the breakpoint id and definition. It is possible to replace this action with any Tcl statement. As an example, the following command defines a breakpoint which prints a descriptive message whenever the variable a is being defined:

```
expect2.1> b -re "^set a " then {
+>    puts "a is being set"
+>    puts "old value of a = $a"
+> }
```

(The first `puts` line carries footnote marker [2].)

When run, it looks like this:

```
expect2.2> c
a is being set
2: set a 2
expect3.3> c
a is being set
old value of a = 2
2: set a 3
expect3.4> c
a is being set
old value of a = 3
2: set a 5
```

Each time the breakpoint occurs, the old and new value of a are displayed. Notice that the first time the breakpoint occurred, a was not defined. In this case, $a was meaningless and the puts command was not executed. If there had been further statements in the breakpoint, they would also have been skipped. Implicit error messages generated by actions are discarded.

Error messages generated in breakpoint expressions are also discarded. It is assumed that such errors are just variables temporarily out of scope.

By default, breakpoints stop execution of the program. It is possible to tell the debugger not to stop by using the commands c, s, n, or r from within an action.

This can be used to trace variables. To illustrate a different effect, the following breakpoint prints out the name of each procedure as it is being defined.

```
expect2.1> b -re "proc (p.)" then
{
+>    puts "proc $dbg(1) defined"
+>    c
```

1. Using braces instead of double quotes permits the previous pattern to be simplified to {\[p4 }. However, the braces prevent the possibility of explicitly matching escaped characters such as \n.

2. Expect prompts with "+>" when an incomplete command has been entered.

```
+> }
0
```

The c command in the last line, allows execution to continue after each breakpoint.

```
expect2.2> c
proc p4 defined
proc p3 defined
proc p2 defined
proc p1 defined
```

The following breakpoint causes the debugger to break after the return of any procedure that has called p4.

```
expect2.1> b -glob "p4" then "r"
```

The following command prints out the string “entering p4” when p4 is invoked. Execution continues for four more steps after that.

```
expect2.2> b -re "^p4 " then {
+>     puts "entering p4"
+>     s 4
+> }
```

Multiple breakpoints can occur on the same line. All corresponding actions are executed. At most one debugger command will be executed, however. For example, if breakpoints trigger commands containing both “s 1” and “s 2”, only the second (or last in general) will have any effect.

Limitations in Breakpoints Actions and Interactive Commands

Debugger commands specified in a breakpoint action occur only after all the breakpoints have completed. For example, the following breakpoint appears to print out the old and new values of every variable about to be set.

```
expect2.1> b -re {^set ([^ ]+) }
then {
+>     puts "old $dbg(1) = [set
$dbg(1)]"
+>     n
+>     puts "new $dbg(1) = [set
$dbg(1)]"
+> }
```

However, the debugger does not actual execute the next procedure call in the program until the breakpoint action completes. This breakpoint therefore prints the old value twice, incorrectly claiming that the latter is the new value.

```
expect4.7> c
old a = 2
new a = 2
```

In this case, it is possible to get the new value by just omitting the last puts. The debugger will then automatically print the new value as part of echoing the next command to be executed.

```
expect4.7>
old a = 2
2: set a 3
```

This example illustrates a limitation of the debugger. The debugger does not use a separate thread of control, and therefore does not allow arbitrary automation of its own commands. For more discussion on these limitations see “Current Limitations and Future Work” on page 15.

General Form of Breakpoints

Expressions and actions may be combined. This follows the syntax of Tcl’s if-then (no “else”). For example, the following command prints the value of $foo whenever it is non-zero.

```
expect2.1> b if {$foo} then {
      puts "foo = $foo"
}
```

The general form of the breakpoint command permits up to one location (specified by pattern, or line number and filename), one expression, and one action. They must appear in this order, but are all optional.

If a location is provided or the if-expression doesn’t look like a line number and/or filename, the “if” token may be omitted. If an if-expression has already appeared, the “then” token is also optional. For example, the following commands have the same effect:

```
expect2.1> b if {$foo} then {
+>     puts "foo = $foo"
+>}
0
expect2.2> b {$foo} {
+>     puts "foo = $foo"
+>}
1
```

When the first argument resembles both a line number and expression, it is assumed to be a line number. The following command breaks on line 17:

```
expect2.3> b 17
2
```

Listing Breakpoints

If no arguments are supplied, the b command lists all breakpoints. The following example assumes the previous three breakpoints have been set and creates two more. Notice that breakpoints zero and one are identical.

```
expect2.4> b -re "^p4"
3
expect2.5> b zz.exp:17 if {$foo}
4
expect2.6> b
breakpoint 4: zz.exp:23 if {$foo}
breakpoint 3: -re "^p4" if {^p4}
breakpoint 2: b 17
breakpoint 1: if {$foo} then {
      puts "foo = $foo"
}
breakpoint 0: if {$foo} then {
      puts "foo = $foo"
}
```

Each breakpoint is identified by an integer. For example, breakpoint four occurs if $foo is true just before line 23 is executed in file zz.exp.

When multiple breakpoints occur on the same line, the actions are executed in the order that they are listed by the b command.

Deleting Breakpoints

A breakpoint can be deleted with the command "b -#" where # is the breakpoint number. The following command deletes breakpoint 4.

```
expect2.7> b -4
```

All breakpoints may be deleted by omitting the number. For example:

```
expect2.8> b -
```

The "h" command – "Help"

The *h* command prints a short listing of debugger commands, arguments and other helpful information.

Changing Program Behavior

When the debugger is active, the variable dbg is defined in the global scope. When the debugger is not active, dbg is not defined. This allows Tcl applications to behave differently when the debugger is running.

Changing Debugger Behavior

By default, long commands are truncated so that the debugger can fit them on a line. This occurs when the debugger prints out a command to be executed and also in the listing from the w command.

The w command has a -width flag which can change the current printing width. It takes a new width as an argument. For example to display long commands (such as procedure definitions):

```
expect2.2> w -w 300
```

Because of the parameter substitutions, the w command may try to display extremely long lines. Imagine the following script:

```
puts [exec cat /etc/passwd]
```

When the debugger is run, w command output will be truncated unless the printing width is quite large.

```
2: exec cat /etc/passwd
expect3.1> s
1: puts [exec cat /etc/passwd]
expect2.2> w
*0: expect -D 1 debug-test3.exp
 1: puts {root:Xu.VjBHD/xM7E:0:1:
Operator:/:/bin/csh
nobody:*:65534:65534::/...
expect2.3> w -w 200
expect2.4> w
*0: expect -D 1 debug-test3.exp
 1: puts {root:Xu.VjBHD/xM7E:0:1:
Operator:/:/bin/csh
nobody:*:65534:65534::/:
daemon:*:1:1::/:
```

```
sys:*:2:2::/:/bin/csh
bin:*:3:3::/bin:
uucp:*:4:8::/var/spool/uucppubli
c:
news:*:6:6::/var/spool/news:/
bin...
expect2.5>
```

When output is truncated, an ellipsis is appended to the end. The default width is 75 which allows some space at the beginning of the line for the procedure call depth information.

By default, no other output formatting is performed. But even short statements can cause lots of scrolling. The following declaration of p4 is less then 75 characters but still takes several lines.

```
% expect -D 1 debug-test.exp
set b 1
expect2.1> s
1: proc p4 {} {
        return [
                expr 5+[expr 1+$x]]
}
```

The -compress flag with argument 1 tells the debugger to display control characters using escape sequences. For example:

```
expect2.2> w -c 1
expect2.3> w
*0: expect -D 1 debug-test.exp
 1: proc p4 {x} {\n\treturn [\n\t
expr 5+[expr 1+$x]]\n}
```

The compressed output is useful for preventing excessive scrolling, and also for displaying the precise characters that should be used in order to match patterns in breakpoints.

To revert to uncompressed output, use the same flag with value 0.

```
expect2.4> w -c 0
```

With no value specified, flags to the w command print out the current value.

```
expect2.5> w -c
0
expect2.6> w -w
75
```

View of the Application Programmer

This section describes how to incorporate the debugger into a Tcl application.

1) Include the file *Dbg.h* in any source that makes calls to the debugger.

```
#include "Dbg.h"
```

2) To start the debugger, call *Dbg_On*. This does not have to be called at program start but can be called at any time.[1]

```
void
Dbg_On(
        Tcl_Interp *interp,
        int immediate);
```

If the “immediate” parameter is 1, the debugger begins interacting with the user immediately. Otherwise, the debugger waits until a new command is about to be executed by Tcl_Eval. Forcing the debugger to begin interacting immediately is useful in slow systems calls such as “read”.

Typical places to call Dbg_On are:

SIGINT Handler

By invoking Dbg_On on receipt of a signal, the user can gain control at any time during program execution.

As an example, Expect enables this using the command-line argument -D 0. Once the debugger is running, SIGINT can still be used to regain control.

Program Start-up

By invoking Dbg_On at program start, the user gains control over the application immediately.

As an example, Expect uses the argument “-D 1” to start this way.

By Application Command

Dbg_On may be called by an application command. A script may then start the debugger

1. Dbg_On should not be called directly from a signal handler but indirectly through Tcl’s signal handling mechanism.

interaction when the command appears in the script.

The debugger does not create a command name association because it has to exist in order to invoke the debugger. Also, because this is the one command name that will always be present in the application, choosing it is best done by the application writer. An example definition might be the name "debugger", using the arguments 0 and 1 similarly to the -D flag, described earlier. But all sorts of other behavior could conceivably be envisioned.

Dbg_Off disables any activity by the debugger. All debugger command names and variables are removed from the interpreter. Dbg_On may be called repeatedly without error before calling Dbg_Off.

```
void
Dbg_Off(
        Tcl_Interp *interp);
```

Dbg_Active returns 1 or 0 depending on whether the debugger is on or off.

```
int
Dbg_Active(
        Tcl_Interp *interp);
```

Several functions are available to customize the debugger. They are described below.

Dbg_ArgcArgv informs the debugger of the command line used to invoke the application. It is used to display the first line of the stack. If the "copy" parameter is 1, the argv array will be copied to a new area of memory. This is useful with applications (e.g., Tk) which modify the argv array.

```
char **
Dbg_ArgcArgv(
        int argc,
        char *argv[],
        int copy);
```

A pointer to the new memory is returned so that it can be freed when the debugger is no longer in use. The individual elements are not reallocated and should not be freed. 0 is returned if no memory is allocated.

Dbg_Interactor names a function that will be called by the debugger to interact with the user.

```
Dbg_InterProc *
Dbg_Interactor(
        Tcl_Interp interp,
        Dbg_InterProc *interactor)
```

Dbg_Interactor allows the debugger to have the same look and feel as that of the application itself. For instance, Expect uses its own interactor by calling:

```
Dbg_Interactor(
        interp,
        exp_interact);
```

Dbg_InterProc is defined as:

```
typedef int (Dbg_InterProc)
        (Tcl_Interp *interp);
```

If an application has no interactor, a very simple interactor (similar to that in tclTest) is provided automatically. The default interactor reads its input from the standard input.

Interactors should prompt for new commands and evaluate them. If commands return TCL_OK or TCL_ERROR, the interactor should simply reprompt for more commands. (Frequent user errors should be expected during interaction.) If commands return TCL_RETURN, the interactor should return TCL_OK. The behavior for commands which return other return values is undefined.

Dbg_Interactor returns the previous definition of its interactor argument.

Dbg_IgnoreFuncs names a function that will be called by the debugger to decide what functions should be ignored.

```
Dbg_IgnoreFuncsProc
Dbg_IgnoreFuncs(
        Tcl_Interp *interp,
        Dbg_IgnoreFuncsProc
                *ignoreproc);
```

Dbg_IgnoreFuncsProc is defined as:

```
typedef int(Dbg_IgnoreFuncsProc)
        (Tcl_Interp *interp,
         char *funcname);
```

If funcname should be ignored, (*ignoreproc)(funcname) should return 1, otherwise it should return 0.

For instance, Expect evaluates the Tcl procedure "prompt1" each time it prepares to prompt the user and "prompt2" if the user has entered a partial command. Expect's procedure to ignore functions is defined as:

```
static int
ignore_procs(char *s)
{
        return(
                (s[0] == 'p') &&
                (s[1] == 'r') &&
                (s[2] == 'o') &&
                (s[3] == 'm') &&
                (s[4] == 'p') &&
                (s[5] == 't') &&
                ((s[6] == '1') ||
                 (s[6] == '2')) &&
                (s[7] == '\0'));
}
```

While avoiding strcmp may be excessive, this function should nonetheless be written efficiently since it is called very frequently.

Implementation

This section describes some of the more interesting parts of the debugger. It is not necessary to read this in order to use the debugger effectively.

The debugger is approximately 1100 lines[1] of C (no Tcl) which compiles to 13K on a Sun 4. The debugger is portable to any platform that already has Tcl. The debugger requires no changes to the Tcl core although it does require access to the Tcl internals.

The debugger uses Tcl's trace facility to get control before execution of every procedure. Tcl temporarily passes control to the function debugger_trap. This function determines whether actions should occur, whether user interaction should occur, and whether execution should continue.

1. The code has a dearth of comments, hopefully somewhat ameliorated by this paper.

The debugger_trap function can be thought of as three distinct parts that nonetheless work together very closely:

prelude: Determine if the debugger should interact with the user, or return to continue execution of the current function.

interactor: Let the user interact with the Tcl environment.

postlude: Manipulate the environment, perhaps letting the user interact again, or returning.

In more detail, the *prelude*'s primary job is to return control if possible, continuing execution of the current command. It attempts to do this as quickly and efficiently as possible, since this code is called upon every trap whether or not it inevitably leads to user-interaction.

For example, there is no reason to trap on debugger commands. They have to be executed in order for the debugger itself to run. But Tcl provides no selective trap mechanism, so the prelude simply has to return in such cases.[2] The prelude also returns if the command about to be executed is interactive (i.e., executed from the debugger interaction) or a function that the application has requested be ignored.

The prelude next evaluates all of the breakpoints. If all breakpoints are unsuccessful or all successful breakpoints have actions, the prelude returns.

The prelude then checks if the previous user command was n, s, c, or r. If so, the environment is examined to see if the requested number of steps occurred, the requested stack level was reached, etc. If the request was not satisfied, the prelude returns.

2. This is not a criticism of Tcl. There is no point in Tcl providing such functionality since it would require a callback to a user routine anyway. This two part decision and execution process is more efficiently performed by combining them into one function, as in debugger_trap here.

If the prelude has not returned at this point, the next phase of the debugger_trap function is entered: the interactor.

The *interactor* executes debugger commands as well as all other Tcl and application commands. Very little special processing is performed. Commands are passed directly to **`Tcl_Eval`**. If the command returns **`TCL_ERROR`**, the interactor ignores it, and continues interacting with the user. It is expected that users will make mistakes while interactively typing commands.

The debugger commands themselves are simple. They record their arguments and return. This may seem surprising, but the fact is that the commands all execute in the wrong context. The current context is the debugger command, and depending on how the command was originally invoked by the user, can be arbitrarily deeply nested beyond the next command in the application to be executed. Getting back to the correct context from a debugger command is very indirect and complicated.

In order for the debugger to get back to the right context, the interactor exits after each debugger command. This is forced by having the debugger commands end by returning TCL_RETURN rather than TCL_OK. The interactor then passes control to the postlude.

The *postlude* processes any requests made by the user while in the interactor. After processing, the interactor is recalled for more user commands, or **`debugger_trap`** returns entirely.

For example, the **`u`** and **`d`** commands set the desired scope and then pass control back to the interactor. The execution scope is later restored before **`debugger_trap`** returns.

Accessing Tcl Internals and Other Problems

The debugger uses Tcl's documented interfaces whenever possible. Undocumented interfaces were used or built in a few cases. This section describes these and other problems encountered while writing the debugger.

Scopes

Implementation of the u and d commands required the ability to arbitrarily walk up or down the procedure call stack. Initially implemented with upvar, this ultimately proved too unwieldy.

Tcl provides support for searching up the procedure call stack through an internal function called TclGetFrame. A new function was constructed to search in both directions. Called TclGetFrame2, it takes additional arguments describing where the true stack boundaries are, as well as where the current scope appears to be.

Several debugger commands take advantage of the ability to directly access the scope. For example, the r command continues execution until the parent scope is encountered. This could probably be implemented with a call to "info level", but the temptation to just compare a single point was too great.

In general, calls to Tcl_Eval are shunned, partly for efficiency but also because they modify the stack. The only calls by the debugger to Tcl_Eval occur when evaluating breakpoints.

Line Numbers

As of version 6.7, Tcl maintains no association between commands and file names and/or line numbers. In retrospect, this is clearly an oversight. Two possibilities seem likely:

Perhaps Tcl originally was never imagined as being used in applications to such an extent that line numbers would be necessary to debugging. Or perhaps, implementing line and file associations is too painful given Tcl's philosophy "a command is a string". Applications may have to provide significant effort to the Tcl core in order for Tcl to support this.

While the current implementation of Tcl currently lacks line number support, the debugger provides all the support for it (see "Current Limitations and Future Work" on page 15).

Commands and Arguments

Displaying the procedure call stack (via the w command) is tricky in two respects.

Tcl does not explicitly keep the original representation of a command while the command is in execution. There is no reason for it to do so. If the command is in a loop, for instance, it will simply be derived again. Without a great deal of work, it is therefore impossible to print out a stack of procedure calls this way. Thus, Expect prints out the values of each argument which is all Tcl has.

In contrast, Expect prints out the original representation whenever the debugger stops and begins the user interaction. Both representations are available for the next command to be executed, but the original representation is used partly because the user can always see the other by entering the w command, and partly because the breakpoint pattern matching makes more sense using the original source code.

Another problem is that Tcl strips off braces and quotes while converting a command to its argv/argc representation. Again, this is reasonable, as the braces and quotes are not formally part of the arguments. The debugger, however, wants to show the user the original code, or perhaps, something that is at least legal.

Without the quotes, simply appending the arguments together is insufficient. For example, a null list will not show up at all. Unfortunately the original information (precisely how it was quoted, for example) simply isn't present. With sufficient time and effort, it coule be reconstructed. However, since the output is destined only for user viewing, the debugger can afford to err in cases that users are unlikely to notice. The debugger uses heuristics for reconstructing the program output.

Other Debuggers

This debugger can be compared to Karl Lehenbauer's debugger [5] (from hereon referred to as "KD"). Besides KD and the debugger described in this paper (from hereon referred to as "DD") no other Tcl debuggers have been constructed.

This comparison will be brief because KD was never completed although it is functional. When it first appeared, it included the proviso that it was "the first cut of an experimental debugger" and "a dim shadow of what is possible". Nonetheless, KD is very interesting because of its differences from DD. The significant differences are as follows:

KD uses two modes: one for debugger commands and one for application commands. This avoids clashes between debugger commands and application commands. All of the commands in KD have, nonetheless, one and two character names.

KD redefines the depth bound of Tcl's trap handler to achieve certain effects such as stepping over procedure calls. DD blindly steps through all calls simulating the same effect by repeatedly checking the current frame pointer. This is, unfortunately, a requirement in order to evaluate breakpoints below the depth bound.

KD lacks the sophisticated breakpoint commands DD offers, but this is just a matter of work. On the other hand, achieving DD's scope manipulation functions may well be impossible through Tcl user-level functions.

Perhaps the most impressive aspect of KD is that only a tiny fraction is written in C. The KD debugger commands are written in Tcl. Users can add new debugger commands or modify the existing commands which are simply stored in a Tcl library. Because of this approach, KD is much slower than a pure C approach. For every user Tcl command, KD executes several debugger Tcl commands.

But speed by itself is not sufficient to justify writing so much of the debugger in C. The real penalty is in the complexity of manipulating an environment at the same time it is being used by the commands manipulating it. Nonetheless, being able to write or customize the debugger commands and functionality with Tcl commands is very interesting and worth pursuing further.

Performance

A thorough study of the performance of the debugger has not been done. Nonetheless, some observations can be made.

Memory

The static size of the debugger has already been stated (see "Implementation" on page 10). When running, the debugger does not significantly increase the in-memory size of a process. There are no symbol tables or other debugging information that has to be loaded. The debugger uses nothing beyond what Tcl already provides to a process not being debugged.

The debugger allocates memory for breakpoints and output buffers, but this is minimal. The result is that the debugger adds approximately 1 to 2% to the size of an application. This is a far cry from debuggers for compiled code, such as gdb and dbx, which typically add 100 to 200%.

Time

The debugger attempts to operate as efficiently as possible in the Tcl framework. Breakpoint evaluation is clearly the most expensive part of the debugger. For example, expression evaluation can require numerous calls to Tcl_Eval.

The time taken to test breakpoints is governed primarily by the complexity and number of breakpoint expressions. Tests of simple expressions ("set a 0" in a loop) suggest that the debugger can add up to 10% to execution time even with no breakpoints. With slower commands (trig functions, system calls, etc.), the execution time overhead drops to an insignificant fraction of the total time. With extensive breakpoint use, the overhead can rise dramatically.

In a debugging session, it is not necessary to have the debugger running all the time. It is possible to enable the debugger only when it is needed, and disable it when it is no longer needed or until needed again. When the debugger is not enabled, it uses no time whatsoever. This technique can help reduce the impact on runtime for some types of debugging.

Current Limitations and Future Work

Experience will undoubtedly prompt many changes and enhancements. This section describes several things that are already contemplated.

The support for defining breakpoints by pattern matching was originally motivated by Tcl's lack of line numbers and filename. While pattern matching is not unique to debuggers (for instance, gdb offers a similar capability although only on function names), this debugger is the first to depend on it to such a great extent. The power of pattern matching is sufficient that many other traditional breakpoint specifications (for instance, by command name) are not necessary.

If line number and filename support is added to Tcl, the debugger stands ready to use it. Currently, the b command parses and records the information, followed by a message that it is unsupported. Line numbers and filenames are also a requirement for a screen-oriented version of the debugger.

The problem of command name clashes between applications and Tcl is long-standing. The debugger deals with this problem by avoidance. The debugger preemptively uses very short command names. (No action is taken to avoid overriding application commands.) The debugger also uses a very small number of command names, overloading them within reason. In contrast, many traditional debuggers define hundreds of commands. While this debugger gains leverage from the existing Tcl commands, this aspect of the design should be studied at more length.

The debugger interface is designed so that multiple debuggers can be used, one per Tcl_Interp. The current implementation, however, does not entirely support this. A handful of static variables are currently shared between all debuggers. For instance, a single linked list of breakpoints is maintained. Differentiating between different Tcl interpreters could potentially be performed by the debugger, but it would be much simpler and more efficient to rely on the Tcl_Interp structure for storage.

An alternative debugger design would move debugger control into a completely separate interpreter. This would enable the ability to write loops or sequences involving multiple debugger commands such as "s;n" which cannot be performed in the current implementation. It is possible to achieve this same ability currently by using an Expect script in a separate process but the result is not as efficient as a single process debugger.

While the debugger can be used with graphic applications (e.g., Tk applications), the debugger is currently intended only to interact with the user in a dumb terminal window. It should be possible to use Expectk [6] to write a GUI for the debugger using only Tk and Expect commands. Building a GUI-based debugger without using Expectk or one of the Expect libraries may require significant rewriting to remove the emphasis on line-oriented interaction.

In the Tk environment, access to the send command opens new possibilities for debugging. Several browsers have already been written. These browsers allow Tcl variables and procedures to be examined and changed without stopping the application. This style of debugging could be combined with the debugger described in this paper.

Conclusion

While the current implementation of Tcl lacks debugger support in some areas, it provides enough hooks to address the most difficult problems in building a debugger.

This paper has described a debugger for Tcl applications. The philosophy of its design is to be as simple as possible by introducing only a few new commands and concepts while using Tcl and application commands as leverage. The result is a reasonably functional and highly-integrated debugger for Tcl applications that is small, fast, and easy to learn and use.

Nonetheless, the possibilities for much more sophisticated debuggers are obvious, and this work can provide a starting point for future endeavors.

Availability

Since the design and implementation of this software was paid for by the U.S. government, it is in the public domain. However, the author and NIST would appreciate credit if this software, documentation, ideas, or portions of them are used.

The debugger may be **`ftp`**'d as **`pub/expect/tcl-debug.tar.Z`**[1] from **`ftp.cme.nist.gov`**. The software will be mailed to you if you send the mail message "**`send pub/expect/tcl-debug.tar.Z`**" (without quotes) to **`library@cme.nist.gov`**.

Acknowledgments

Thanks to John Ousterhout, Sarah Wallace, Susan Mulroney, Bob Bagwill, and Rob Savoye for critiquing this work, and providing suggestions that greatly enhanced the usability of the debugger and readability of the paper.

The author gratefully acknowledges John Ousterhout for creating Tcl. Not only does Tcl solve a significant problem in software design, but the code itself as well as the documentation are comprehensive and written with consummate style. Tcl is truly a pleasure to use.

Portions of this work were funded by the NIST Scientific and Technical Research Services as part of the ARPA Persistent Object Base project, and the Computer-aided Acquisition and Logistic Support (CALS) program of the Office of the Secretary of Defense.

Disclaimers

Trade names and company products are mentioned in the text in order to adequately specify experimental procedures and equipment used. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the products are necessarily the best available for the purpose.

1. The ".Z" file is compressed. A ".z" version is also available which is gzipped.

References

[1] Ousterhout, John, “Tcl: An Embeddable Command Language”, *Proceedings of the Winter 1990 USENIX Conference*, Washington, D.C., January 22-26, 1990.

[2] Ousterhout, John, “Tcl(3) – Overview of Tool Command Language Facilities”, *unpublished manual page*, University of California at Berkeley, January 1990.

[3] Libes, Don, “Expect: Curing Those Uncontrollable Fits of Interaction”, *Proceedings of the Summer 1990 USENIX Conference*, pp. 183-192, Anaheim, CA, June 11-15, 1990.

[4] Libes, Don, “Expect: Scripts for Controlling Interactive Programs”, *Computing Systems*, pp. 99-126, Vol. 4, No. 2, University of California Press Journals, CA, Spring 1991.

[5] Lehenbauer, Karl, “A Source Level Debugger for Tcl”, *Usenet* Message-ID: <1992Jan03.220658.22059@NeoSoft.com>, January 3, 1992.

[6] Libes, Don, “Expectk”, *unpublished manual page*, National Institute of Standards and Technology, January 1993.