SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 27(2), 123–137 (FEBRUARY 1997)

Automation and Testing of Character-graphic Programs

DON LIBES
National Institute of Standards and Technology, Bldg 220, A-127, Gaithersburg, MD, 20899, U.S.A.
(email: libes@nist.gov)

SUMMARY

This paper describes a technique that allows automation and testing of character-graphic programs using existing public-domain tools. Specifically, Tcl, Tk, and Expect are augmented with a terminal emulator in order to build a screen representation in memory. This screen can be queried in a high-level way and the interaction can be further controlled based on the screen representation.

One immediate use of this is to build a test suite for automating standards conformance of the interactive programs in POSIX 1003·2 (Interactive Shells and Utilities). This technique is portable and inexpensive. All the software described in this paper is free or in the public domain. © 1997 by John Wiley & Sons, Ltd.

KEY WORDS: conformance testing; Expect; interaction automation; POSIX 1003.2; regression testing; Tcl/Tk

INTRODUCTION

This paper describes a general technique that allows automation and testing of character-graphic programs using portable and inexpensive tools. Specifically, Tcl, Tk, and Expect are augmented with a terminal emulator in order to build a screen representation in memory. This screen can be queried in a high-level way and the interaction can be further controlled based on the screen representation.

One immediate use of this is to build a test suite for automating standards conformance of the interactive programs in POSIX 1003.2 (Interactive Shells and Tools).[1]

BACKGROUND

Tcl (Tool Command Language) is an embeddable language library that can be linked to other applications. Tcl provides a fairly generic but reasonably high-level language. The language is interpreted and resembles the UNIX shell in many ways. Elements are also derived from C and LISP. Despite its mixed heritage, much of the excess baggage from these other languages has been omitted, leaving a modest but capable language.

Tcl is extensible. Two popular Tcl extensions are Tk and Expect. Tk enables control of graphic user interfaces. Expect enables control of interactive character-

CCC 0038–0644/97/020123–15

Received 25 October 1995
Revised 19 March 1996

oriented interfaces. Both Tk and Expect can work together. For example, they can be used to layer a graphic user interface on top of an existing character-oriented program.[2]

Tcl and Tk are described by Ousterhout.[3] Expect is described by Libes.[4,5] The remainder of the paper assumes a reasonable understanding of Expect, Tcl, and Tk.

Expect processing in non-character-graphic programs

In non-character-graphic applications, characters are written on each line from left to right. After completing a line, characters are written to the next line. When the last line of the screen is filled, the screen is scrolled. The oldest line at the top of the screen is deleted, all the other lines are moved up, and new characters are written to the new line at the bottom of the screen.

Because characters appear in exactly the order that they are written, it is simple to wait for specific patterns. As characters arrive, they are appended to a buffer. The buffer can then be searched for the patterns of interest.

For example, suppose a program prompts with the string '**yes or no:**'. This prompt can be detected by waiting for exactly that string to appear in the output of the program.

Expect is a popular public-domain program that automates interactive programs. Using Expect, the actual command to wait for the string '**yes or no:**' is:

```
expect "yes or no:"
```

Expect has a rich set of built-in tools to describe patterns. However, they are all serial in nature. Expect sees a stream of characters and does not attempt to interpret the characters in a different order than they were received.

Expect processing in character-graphic programs

In contrast to non-character-graphic programs, character-graphic programs write characters to arbitrary character locations on a screen or window. For example, a DEC VT100 terminal can display a 24 by 80 grid of printable ASCII characters. Characters can only appear in discrete locations in the grid. However, the grid can be filled in any order and characters at any location may be replaced at any time by other characters.

Special character sequences, usually beginning with an escape character (ASCII ESC), are used to position subsequent characters in the grid. These sequences are referred to as *positioning sequences*.

Because the grid may be filled in any order, it is not trivial to watch a stream of characters for patterns. Typically, such programs take advantage of characters that already exist on the screen to reduce the amount of characters that have to be produced to update the display.

For example, suppose a line on the screen contains '**yesterday.**'. If this is to be replaced with the '**yes or no:**' prompt, the program can rewrite the entire line with '**yes or no:**'. However, the program can achieve the same effect by replacing the '**te**' with '**o**' and '**day.**' with '**no:**'. This is shown in Figure 1.

The output of this program to produce '**yes or no**' would be:

yesterday. <*positioning sequence*> **o** <*positioning sequence*> **no:**

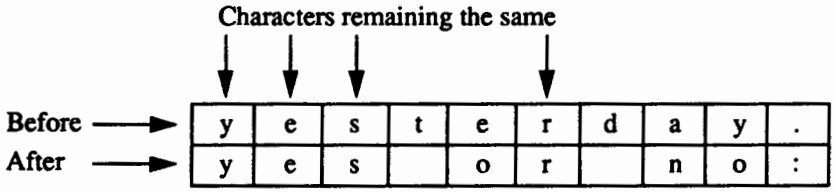

Figure 1. Replacing selected characters on a screen

The simple Expect command used earlier would not be able to match '**yes or no**' in such output. However, with an understanding of how to interpret the positioning sequences, it is possible to model the screen and match the string. In that case, the match is not made against the output directly. Instead, the match is made against the model of the screen.

TERMINAL EMULATON

A screen may be modelled using *emulation*. Indeed, emulation is the basis for terminal emulators. Terminal emulators create a model of the screen and display it on a windowed system such as the X Window System (X11).[6] However, terminal emulators are not designed to support detection of patterns on the screen.

In this section, a terminal emulator is presented that provides a framework upon which to perform screen analysis. Functionally, the emulator is capable of supporting sophisticated character-graphic programs such as Vi and Emacs.[7,8] Hooks are provided so that screen analysis can be done after each screen update.

There are many ways to maintain a grid of characters. For simplicity, we started out using Tk, a system for controlling X graphics. A Tk text widget is convenient for maintaining the grid because Tk can automatically display the grid in a window and the Tk understands typical terminal features such as highlighting. In practice, it is not necessary to display the grid. Indeed, non-display of the grid is useful when automating an existing program. In many cases, user comprehension of the character-graphic user interface is no longer necessary so there is no need to display it. A version of this work was repeated without using Tk—a simple Tcl array was used to model the screen.

To run processes, a shell is used through which any programs under test are invoked. By using Expect's **spawn** command, a pseudo-terminal is automatically supplied, allowing applications to believe that they are talking to a real user. Without the pseudo-terminal, many nominally character-graphic applications would forego their character-graphic interface, making them impossible to test. Many applications would refuse to run at all.

In theory, the task of understanding screen manipulation sequences is straightforward. However, in reality, it is complex. Some of the problems are:

- Many vendors use non-standard screen manipulation sequences.
- Even with a single screen-manipulation definition, there is an infinite number of sequences that can generate a particular screen image.
- High-level databases and libraries exist to deal with the multi-vendor problem, however there is no single standard.

- Some programs do not follow the specifications described by the high-level databases/libraries.

Intuitively, the way to build a terminal emulator is to figure out what the character sequences mean and model this in computer code. The solution presented in this paper is not far from that idea, but it gets there by a somewhat circuitous route.

First, it is necessary to understand that there is no standard terminal type. Although there is an ANSI standard, it is so limited that all vendors extend it. Naturally, these extensions are rarely compatible with one another. Indeed, manufacturers often produce extensions that are different even within their own model lines.

Several attempts have been made to define high-level databases and software interfaces to understand these hundreds of definitions. However, these interfaces are for producing character sequences, not consuming them.

Given an arbitrary character sequence, there is no trivial way to figure out what it does. Presuming a particular terminal type simplifies the problem but does not necessarily make it solvable. Inverting database descriptions may not be possible if any of the sequences are identical. For instance, consider descriptions that use the same sequence for both highlight and inverse. In this case, there is no way of telling which one was intended just by knowing the sequence had arrived. A related problem exists with individual sequences that are identical to a sequence of other sequences. A different problem arises when terminals are used beyond their documented limits. In some cases, sequences defined with only enough space for 24 rows can match two different requests if a terminal with more than 24 rows is emulated. This is a common scenario with emulated terminals where users expand the terminals many times larger than their physical counterparts. In such cases, which request is correct can only be determined by the undocumented operation of the physical terminal itself. This can change from one release to the next and is not necessarily derivable via software.

To avoid these problems, a theoretically 'ideal' terminal was designed. Designing a terminal from scratch makes it possible to avoid the difficulties of having to deduce the characteristics of another existing terminal. However, there are two drawbacks of an ideal terminal:

- An ideal terminal cannot be automatically displayed on a terminal emulating a different type.
- A program that only generates output for a specific type will not necessarily display correctly on an ideal terminal.

Fortunately, both of these are moot. The first drawback is irrelevant partly because typically the emulator itself provides a display. This display process is described later. In addition, the emulator can be augmented to consume characters meant for one terminal type and convert this into characters to drive yet another type. The second drawback is irrelevant because the programs of interest should not be tied to a particular terminal type but should be terminal independent. Whereas it is possible to force the emulator to understand a particular terminal type, it can be much more difficult because the ideal terminal is invariably much simpler than any real terminal.

Defining terminal definitions

An arbitrary terminal definition would be meaningless if there were no way to inform programs of it, but the same databases as before serve this purpose. The approach taken by modern databases is to support arbitrary terminal types through the use of a terminal description language. Unfortunately, there is no single standard.

In UNIX environments, there are two 'standards'—Termcap and Terminfo.[9] The presence of one of these can often be explained by the derivation of the system. Termcap was invented as Berkeley and can be found on Berkeley-derived systems. Terminfo was a redesign provided by AT&T and can be found on AT&T (i.e. SV) derived systems. Many systems support both and it is not uncommon to find half the utilities on the system using Termcap and half using Terminfo. Hence, the solution in this paper necessarily implements both. The script is forgiving in that it runs even if one of the two implementations is absent.

Fortunately, it is much easier to design a terminal description from scratch than it is to mimic an existing terminal description. The reason is that few sequences are actually mandatory. For instance, relative cursor motion can be simulated with absolute cursor motion. This one observation alone dramatically simplifies descriptions because there are often dozens of relative cursor motions that can be replaced by a single absolute cursor motion definition. Using a single, albeit more complex, definition also turns out to be more efficient than many relative cursor motion operations. The explanation for this efficiency is described later.

The following code establishes descriptions in both Termcap and Terminfo style using the ideal terminal type, arbitrarily named '**tk**'. The code succeeds even if Termcap and Terminfo are not supported on the system. This code actually has to be executed before the **spawn** shown earlier in order for the environment variables to be inherited by the process.

The Termcap and Terminfo definitions are very similar so only the Termcap definition is described here. The definition is made up of several capabilities. Each *capability* describes one feature of the terminal. A capability is expressed in the form **xx=value**, where **xx** is a capability label and **value** is the actual string that the emulator receives. For instance the **up** capability moves the cursor up one line. Its value is the sequence: escape, '**[**', '**A**'. These sequences are not interpreted at all by Tcl so they may look peculiar. The complicated-looking sequence (**cm**) performs absolute cursor motion. The row and column are substituted for each *%d* before it is transmitted. The character string '**\E**' is replaced with a true escape character. The remaining capabilities are *non-destructive space* (**nd**), *clear screen* (**cl**), *down one line* (**do**), *begin standout mode* (**so**) and *end standout mode* (**se**). The actual definitions are based on the ANSI terminal definition.[10] This is a purely arbitrary choice.

```
set env(TERM) "tk"
set env(TERMCAP) {tk:
    :cm=\E[%d; %dH:
    :up=\E[A:
    :nd=\E[C:
    :cl=\E[H\E[J:
    :do=^J:
    :so=\E[7m:
```

```
    :se=\E[m:
}

set env(TERMINFO) /tmp
set ttsrc "/tmp/tk.src"
set file [open $tksrc w]

puts $file {tk,
    cup=\E[%p1%d;%p2%dH,
    cuu1=\E[A,
    cuf1=\E[C,
    clear=\E[H\E[J,
    ind=\n,
    cr=\r,
    smso=\E[7m,
    rmso=\E[m,
}
close $file
catch {exec tic $tksrc}
exec rm $tksrc
```

A generic standout mode is used for brevity in this paper. Extending it to specific ones such as underlining and highlighting is straightforward.

Maintaining and querying the terminal display

The text widget maintains the terminal display internally. Most of the details are not relevant to this paper and names such as **term_init** and **term_clear** should be intuitively obvious. One procedure will be described in more detail to provide a taste for the implementation and in order to understand some of the problems encountered.

The **term_down** procedure moves the cursor down one line. If the cursor is already at the end of the screen, the text widget appears to scroll. This is accomplished by deleting the first line and then creating a new one at the end.

```
proc term_down {} {
    global cur_row rows cols term

    if {$cur_row < $rows} {
        incur cur_row
    } else {
        # already at last line of term, so scroll screen up
        $term delete 1.0 "1.end + 1 chars"

        # recreate line at end
        $term insert end [format %*s $cols ""]\n
    }
}
```

There is no correspondingly complex routine to scroll up because the Termcap/Terminfo libraries never request it. Instead, they simulate it with other capabilities. In fact, the Termcap/Terminfo libraries never request that the cursor

scroll past the bottom line either. However, non-character-graphic programs such as **cat** and **ls** do, so the terminal emulator understands how to handle this case.

The **term_insert** procedure writes a string to the current location on the screen. Due to the nature of Tk's text widgets, the procedure does its work by first deleting the existing characters and then inserting the new characters. This is a good example of where Termcap/Terminfo fail to have the ability to adequately describe a terminal. The text widget is essentially always in 'insert' mode but Termcap/Terminfo have no way of describing this.

One capability of which the script cannot take advantage, is that Termcap/Terminfo can be told not to write across line boundaries. Again however, programs such as **cat** and **ls** expect to be able to write over line boundaries.

At the very end of **term_insert** is a call to **term_chars_changed**. This is a user-defined procedure called whenever visible characters have changed. For example, the following code finds when the string **foo** appears on line 4 column 7:

```
if {[string match foo* [$term get 4.7 4.end]]}
```

The following code tests if character at row 4 col 5 is in standout mode

```
if {-1 != [lsearch [$term tag names 4.5] standout]} . . .
```

Information can also be retrieved. For example to return the entire screen image:

```
$term get 1.0 end
```

The following example code returns indices of the first string on lines 4 to 6 that is in standout mode

```
$term tag nextrange standout 4.0 6.end
```

The utility procedure **term_update_cursor** is called to update the visible cursor. This procedure calls a user-defined procedure, **term_cursor_changed**. A possible definition might be to test if the cursor is at some specific location:

```
if {$cur_row == 1 && $cur_col == 0} . . .
```

A single expect command suffices to read and parse the sequences. The command has a pattern to match each possible sequence. An abbreviated implementation is shown below. For instance, a non-destructive space sequence causes the current column to be incremented. A carriage-return sets the current column to 0. Notice how simple the code is for absolute cursor motion. It is basically two assignment statements. Because it is so simple, there is no need to supply Termcap/Terminfo with information on relative cursor motion commands. They cannot be substantially faster.*

```
expect_background {
    -re "^\[^\x01-\x1f]+" {     # Text
        term_insert $expect_out (0,string)
        term_update_cursor
    } "^\r" {                   # (cr,) Go to beginning of line
        set cur_col 0
```

* The definition for non-destructive space might be seen as a concession to speed, but in fact it is required by some buggy versions of Termcap that operate incorrectly if the capability not defined. The other relative motion capabilities are assumed by the terminal driver for non-character-graphic tools such as **cat** and **ls**.

```
        term_update_cursor
    } "^\n" {                   # (ind,do) Move cursor down one line
        term_down
        term_update_cursor
    } "^\033\\[C" {             # (cuf1,nd) Nondestructive space
        incr cur_col
        term_update_cursor
    { -re "^\033\\[(\[0-9]*);(\[0-9]*)H" {
                                # (cup,cm) Move to row y col x
        set cur_row [expr $expect_out(1,string)+1]
        set cur_col $expect_out(2,string)
        term_update_cursor
    }
```

Bindings define how the emulator should handle user events such as user keystrokes and mouse motion. For example, the following statement defines a binding that applies to any keypress event. Upon occurrence of such an event, its action sends the corresponding ASCII character to the process. Keypress events that do not have an associated ASCII character, such as 'shift' and 'control', are discarded.

```
bind $term <Any-KeyPress> {
    if {"%A" != ""} {
        exp_send "%A"
    }
}
```

The meta key is simulated by sending an escape character. Most programs understand this convention, and it is convenient because it works over **telnet** links.

These bindings are the same for any terminal and thus are not defined by explicit capabilities. Bindings that are unusual do require capabilities. For example, some terminals have function keys that generate a string of characters, typically unique to a particular brand of terminal. This behaviour is described using a capability. For instance, the capability for function key 1 to send escape, '**O**' , and '**P**' could be described in either of two ways:

```
:k1=\EOP:          Termcap-style
:kf1=\EOP:         Terminfo-style
```

The matching binding is:

```
bind $term <F1> {exp_send "\033OP"}
```

Another event that could be handled is an X configure event, which is generated when the user changes the size of the terminal window. For simplicity, the code shown here does not support this. The actual code is not significant, however it requires additional interfaces. For example, the user should be able to choose between various behaviors such as whether or not characters should be 'forgotten' if the screen is temporarily shortened and then lengthened.

There are not Termcap/Terminfo capabilities to describe this behaviour. There should be such capabilities, but it requires a survey of terminal vendors to see what is actually implemented. The obvious choices are 'blank fill everything' or 'restore

everything'. Yet some tools, such as the ubiquitous xterm program, have mixtures such as 'blank fill when widening' and 'restore when lengthening'.

Termcap/Terminfo are missing many important capabilities. Another example is whether or not to restore the screen to the way it looked prior to the application's execution. Some terminals do it one way, some the other. An emulator can do it either way of course. But because Termcap/Terminfo provide no capability for it, another interface must be provided.

USING THE TERMINAL EMULATOR FOR TESTING AND AUTOMATION

It is possible to use the terminal emulator described in the previous section partly or fully to automate or test character-graphic applications. For instance, each **expect**-like operation could be a loop that repeatedly performs various tests of interest on the text widget contents. In the following code, the entrance to the loop is protected by '**tkwait var test_pats**'. This blocks the loop from proceeding until the **test_pats** variable is changed. The variable is changed by the **term_chars_changed** procedure, invoked whenever the screen changes. Using this idea, the following code waits for a **%** prompt anywhere on the first line:

```
proc term_chars_changed {} {
    uplevel #0 set test_pats 1
}

while 1 {
    if {!$test_pats) {tkwait var test_pats}
    set test_pats 0
    if {[regexp "%" [$term get 1.0 1.end]]} break
}
```

Writing a substantial script this way would be clumsy. Furthermore, it prevents the use of control flow commands in the actions. One solution is to create a procedure that does all of the work handling the semaphore and hiding the **while** loop. Such a procedure is described in the next section.

Term_expect

term_expect is a procedure that simplifies the writing of **expect**-like operations. term_expect replaces the ubiquitious while loop (see above) and the other control machinery. The interface of **term_expect** is similar to the original **expect**. Time-outs, defaults, patterns, and actions are all supported.

A significant difference is that instead of patterns, the user provides executable tests. Thus, **term_expect** is written as a series of test-action pairs:

```
expect_term test1 action1 test2 action2 . . .
```

Because the tests can be arbitrarily large lists of statements, they are grouped with braces. For example, the previous test could be written:

```
expect_term {[regexp "%" [$term get 1.0 1.end]]} {;# no-op}
```

Any number of test-action pairs can be provided. The action can be omitted if empty as is the case here. Actions can contain multiple statements. They can also involve flow control such as break, continue and return.

Tests can contain multiple statements. Because the tests can be arbitrarily large lists of statements, they are grouped with braces. Any non-zero test result causes term_expect to be satisfied, whereupon it executes the associated action. One special test ("**timeout**") is provided to support timeouts, analogous to **expect**.

The implementation of **term_expect** follows the model shown above, using a loop and waiting to be called back when the screen has been updated. The actual code is more complicated because it addresses scoping problems and must handle flow control and timeouts.

Example of partial automation—Rogue

Rogue is an adventure game that presents a player with various physical attributes, such as strength and health. The attributes are displayed using character graphics. For instance "**Str: 16**" indicates a strength of 16. This strength value is the default but the game randomly provides a much better strength of 18. It is provided rarely however, and quitting the game is clumsy enough that players do not repeatedly restart the game in hopes of getting the high strength. In particular, the game is quit by initially pressing '**Q**'. The game then fills in the word '**Quit**' and asks '**Are you sure?**'. The user must answer '**y**', wait for the shell prompt to reappear and then re-enter the name of the game ('**rogue**') to restart it.

One of the earliest examples of Expect was a script that automated this particular interaction, allowing users always to be able to start with optimal initial configurations for the game.[11] However, because the game uses character graphics, the script could conceivably miss the patterns for which it is looking.

Using the **term_expect** procedure described above, it is possible to write a replacement script for Rogue that fully understands the character graphics. For instance, the first test looks for the shell prompt ('**%**') in either the first or second line on the screen. After sending '**rogue**', the script looks for a strength of 16 or 18. If 18 is found, the break action is executed causing the loop to break. A strength of 16 causes the script to terminate the game and restart a new one for examination. The meaning of the rest of the script should be obvious.

```
while 1 {
    term_expect {regexp "%" [$term get 1.0 2.end]}
    exp_send "rogue\r"
    term_expect {regexp "Str: 18" [$term get 24.0 24.end]}
    break \
        {regexp "Str: 16" [$term get 24.0 24.end]} {}
    exp_send "Q"
    term_expect {regexp "quit" [$term get 1.0 1.end]}
    exp_send "y"
}
```

In contrast to the original Rogue script, there is no **interact** command at the end of this one. Because of the bindings, the script is *always* listening to the keyboard! If desired, this implicit interaction can be disabled by removing or overriding the **KeyPress** bindings that appear at the end of the terminal emulator.

Example of total automation—querying a database

The following example connects to the Cornell University Library and makes a number of queries through its menu system. Interestingly, this library expects to drive a 3270 terminal. A 3270 terminal is not like a typical serial terminal and traditional programs such as telnet and rlogin do not support the 3270 interface. Thus, Expect uses the tn3270 program to convert the 3270 interaction to a Curses-style character stream, which can then be handled as usual.

First, the shell prompt is waited for and the 3270 emulator is started.

```
term_expect {regexp {.*[>%]} [$term get 1.0 3.end]}
exp_send "tn3270 notis.library.cornell.edu\r"
```

The next step is to get through the library's login interaction.

```
term_expect {regexp "desk" [$term get 19.0 19.end]} {
                exp_send "\r"
    }
```

Once in the library system, all the menus prompt the same way. This is a common situation and calls for yet higher-level tools than **term_expect**. It is difficult to define such higher-level tools in a way that would be reusable to others. Fortunately, they are almost always short, so it is not difficult to write them anew each time. Here are example utility routines to handle this repetitive situation for the Cornell University Library.

```
proc waitfornext {} {
    global cur_row cur_col term
    term_expect {expr {$cur_col==15 && $cur_row == 24 && \
                    " NEXT COMMAND: " == [$term get 24.0
                    24.16]}} {}
}

proc sendcommand {command} {
    global cur_col
    exp_send $command
    term_expect {expr {$cur_col == 79}} {}
}
```

Now the interactions with the library are trivial. The remaining commands look for a book using the keywords '**sound**' and '**scottish**'. The first book is selected and its long form is displayed. Finally the next page of the long form is shown.

```
waitfornext
sendcommand "k=sound and scottish\r"
waitfornext
sendcommnd "1\r"
waitfornext
sendcommand "1on\r"
waitfor next
sendcommand "for\r"
```

The view of the Tk terminal emulator, after these queries, is shown in Figure 2.

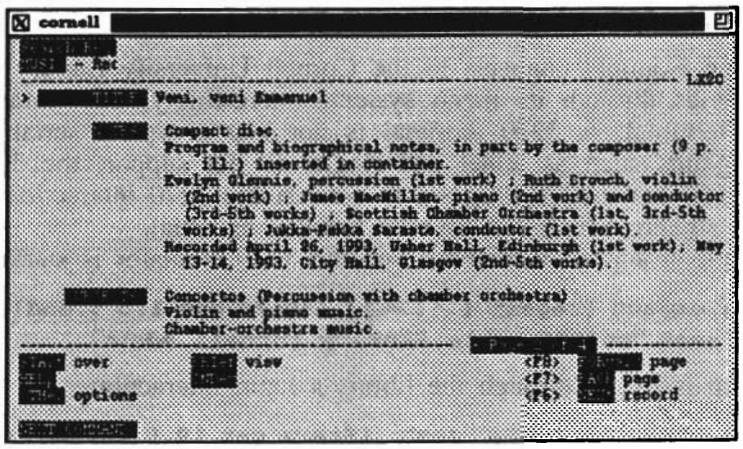

Figure 2. Terminal emulator after several queries to Cornell University Library

Testing

In the automation examples, only the desired outcome was anticipated by the script. Consider the database query example. If the network was down, the connection to Cornell would fail. A new release of the tn3270 program might have a bug in it. Many other problems are possible. Robust scripts must be able to deal with all of these alternative outcomes.

Handling different outcomes is possible by adding additional tests to each call of **term_expect** command as was done in the Rogue example to check for the strength of 16 and 18 simultaneously. Conformance testing, which either succeeds or fails, requires only the desired pattern and the lack of the pattern within a given amount of time. For example, the following script checks for a shell prompt. If found, it prints '**found**'. If not found within 20 seconds, it prints '**not found**'.

```
set timeout 20
term_expect {
    set line [$term get 1.0 2.3nd]
    regexp "%" $line
} {
    puts "found"
} timeout {
    puts "not found"
}
```

The special pattern **timeout** matches when sufficient time has expired, just as the **expect** command does. The associated action is executed as with any successful pattern match. Unkike the **expect** command, a test for end-of-file is not provided because a terminal emulator should not exit just because the applications making use of it do so.

More sophisticated checking can require the addition of many other tests and failure modes. For example, consider testing a character-graphic editor such as Vi. It is not sufficient to look for a particular pattern. Rather, the entire screen must be

correct after an interaction. Doing such a test is straightforward using Tcl's built-in string comparison command:

```
string compare $desired_image [$term get 1.0 24.end]
```

Because tests may execute arbitrary commands, this can also be done using algorithms rather than literal patterns. For example, Vi begins with a screen displaying tildes down the left-most column. The following code tests for this.

```
for {set i 1} {$i<=24} {incr i} {
    if ["~" != [string trimright [$term get $i.0 $i.end]]] {
        return 1
    }
}
return 0
```

Other tests may be useful to handle unusual but possible conditions. For example, when testing Emacs, occasional messages appear such as those relating to garbage collection. Assuming the status line is stored in the variable **status_line**, the unpredictable messages from Emacs could be detected with the following tests:

```
string match "Garbage collecting. . .Done" $status_line
string match "Garbage collecting. . ." $status_line
string match "Auto-saving. . .Done" $status_line
string match "Auto-saving. . ." $status_line
```

A large body of tests and testing expertise has been constructed using Expect.[12,13] As in the Rogue example, it is straightforward to convert **expect**-style tests to **term_expect**-style tests. In many cases, the **term_expect**-style tests are more robust and it is likely that some test suites will be rewritten to take advantage of this additional rigor. However, little actual experience has been collected.

At the same time, **expect_term** shares with **expect** one difficulty of constructing such tests. Namely, there is a tradition of avoiding formal test specifications for user interfaces. As an example, POSIX lacks such test specifications. Thus, test implementation often includes specification of the test as well. For this reason, substantial time must be allotted to designers of test suites for character-graphic interfaces.

Note that the emulator is not a tool for testing Termcap, Terminfo, or other terminal libraries. The emulator defines a very small number of minimal capabilities, exactly the opposite of what is needed to test capability libraries. The emulator necessarily assumes Termcap and/or Terminfo are functioning normally.

ALTERNATIVES

ExpecTerm was earlier work that implemented a universal terminal emulator inside of Expect itself.[14] The emulator was written in C. Access to the terminal emulator was provided by several additional flags to the **spawn** and **expect** commands. For instance, the following **expect** command looked for '**aaaaa**' in the first five columns of the first row and '**bbbbb**' in the first five columns of the second row.

```
expect -rows 0:1 -cols 0:4 "aaaa/nbbbbb"
```

Expecterm provided access to character attributes (reverse, dim, blink, etc.) and

had a variety of other options. For instance **-rrows** allowed relative region specifications and negative integers indexed from the end of the region.

Expecterm suffered exactly the difficulties of a universal terminal emulator mentioned earlier. In particular, because it was not possible to invert Termcap/Terminfo, entries had to be handediting. This was not a task for most users. To ameliorate this, Expecterm came with its own Terminfo file for a few particularly popular terminals. The replacement definitions were subsets guaranteed to be clean of ambiguities, exactly like the Tk definition shown earlier.

The Expecterm command interface pushed the complexity of the screen modelling into Expect itself. The primary disadvantage was that it defined yet a new sub-language that users had to learn. (Users already had to master Expect's original sub-language for expressing patterns in a serial stream.) Because terminal emulator pattern matching was performed by compiled code, when patterns did not match, it could become very difficult for users to figure out what was going wrong. In contast, the **term_expect** approach uses the existing Expect pattern-matching sub-language tied together with existing control flow commands, both of which users are already familiar with. The ability to use control flow commands in tests permits algorithmic tests as in the Vi example.

The Expecterm internals and interface have not continued to be maintained and have not been supported by Expect for four years. However, Expecterm is not being ignored. The **term_expect** approach described in this paper should merely be considered an alternative interface—currently in favour. It remains to be seen whether the **term_expect** procedure is a desirable user interface. Because it is modifiable by the user, it is likely, however, that this will more easily serve as a testbed for better future interfaces.

CONCLUSIONS

A terminal emulator has been described that provides an infrastructure for testing and automation of character-graphic programs. The tool is stable and robust. At the same time, because it is implemented entirely in interpretive Tcl, it is accessible to the user and easily modified should the need for extension or the desire to experiment arise.

Much of the credit for the features and simplicity are due to the supporting tools in which the work was implemented. Specifically, Expect, Tcl, and Tk provide a convenient environment for the implementation of this work. Expect provides a pseudo-terminal to make applications run as if they were directly connected to a user at a real terminal. There is no other tool that provides access to pseudo-terminals trivially and portably. Expect additionally provides convenient commands for pattern matching regular expressions from a stream of characters, much in the style of Lex. This is a good fit to the problem of parsing terminal sequences. Tcl provides a high-level interface for control. Because Expect was originally designed to be controlled by Tcl, it should not be surprising that the combination of the two works well here. Similarly, Tk extends Tcl, with the ability to display X windows. This solves the problem of displaying the results of the emulator and providing data structures for the display. Because there is a well-known tool (Expectk) that provides Tcl, Tk, and Expect together, the triad was an obvious tool to apply to this problem.

Availability

The software described in this paper is freely available. However, the author and NIST would appreciate credit if this software, documentation, ideas, or portions of them are used.

The scripts and programs described in this document may be ftp'd as pub/expect/expect.tar.Z from ftp.cme.nist.gov. The software will be mailed to you if you send the mail message 'send pub/expect/expect.tar.Z' (without quotes) to library@cme.nist.gov.

ACKNOWLEDGMENTS

Much of the development of Expect was funded by the NIST Scientific and Technical Research Services. The Tk-less implementation was done by Adrian Moriano, Cornell University. Adrian also wrote the script to interact with the Cornell University Library. Thanks to Steve Ray, Josh Lubell, Kathy Miles, and several anonymous reviewers for proofreading this paper.

REFERENCES

1. *Portable Operating System Interface (POSIX)—Part 2: Shell and Utilities*, Federal Information Processing Standards Publication 189, National Institute of Standards and Technology, October 11, 1994.
2. D. Libes, 'X wrappers for non-graphic interactive programs', *Proc. Xhibition 94*, San Jose, California, 20–24 June 1994.
3. J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, ISBN 0-201-63337-X, April 1994.
4. D. Libes 'Expect: scripts for controlling interactive programs', *Computing Systems*, **4**(2), 99–126, University of California Press Journals, CA, Spring 1991.
5. D. Libes, *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*, O'Reilly and Associates Inc., 602 pp. ISBN 1-56592–090–2, January 1995.
6. A. Nye, T. O'Reilly *et al. The X Window System Series*, O'Reilly and Associates Inc., Sebastopol, CA.
7. L. Lamb, *Learning the vi Editor*, O'Reilly and Associates Inc., ISBN 0-937175-67-6, October 1990.
8. R, Stallman, *GNU Emacs Manual*, Free Software Foundation Inc., ISBN 1-88211404-3, July 1994.
9. B. Goodheart, *UNIX Curses Explained*, Prentice Hall, 1991.
10. *ANSI X3.64–1979 (R1990)—Additional Controls for Use with the American National Standards Code for Information Interchange*, ANSI, 1990.
11. D. Libes, 'Expect: curing those uncontrollable fits of interaction', *Proc. Summer 1990 USENIX Conference*, Anaheim, CA, 11–15 June 1990, pp. 183–192.
12. D. Libes, 'Regression testing and conformance testing interactive programs', *Proc. Summer 1992 USENIX Conference*, San Antonio, TX, 8–12 June 1992.
13. R. Savoye, 'The Solution: DejaGnu', *Free Software Report*, Mountain View, CA, **3**(1).
14. C. J. Matheus and M. D. Weissman, *expecTerm*, URL:ftp://ftp.aud.alcatel.com/tcl/extensions/expecTerm1.0beta.tar.gz, May 1992.