Regression Testing and Conformance Testing Interactive Programs

Don Libes – National Institute of Standards and Technology

ABSTRACT

Testing interactive programs, by its nature, requires interaction – usually by real people. Such testing is an expensive process and hence rarely done. Some interactive tools can be used non-interactively to a limited extent, and are often tested only this way. Purely interactive programs are rarely tested in any systematic way.

This paper describes testing of interactive line and character-oriented programs via Expect. An immediate use of this is to build a test suite for automating standards conformance of all of the interactive programs in POSIX 1003.2a (interactive shells and tools), something which has not yet been accomplished by any means.

Introduction

Dennis Ritchie said [1] that ''A program designed for inputs from people is usually stressed beyond the breaking point by computer-generated inputs.'' I would add the following: Any program useful to people – interactively – is likely to be useful to programs – non-interactively. A corollary of Ritchie's statement is that correct software function during normal human use is not a very good test of a program's total correctness.

I claim that even when humans are explicitly *testing* interactive software, the results are still quite unreliable. Humans have many drawbacks:

- Humans know what is reasonable, and strive to avoid incorrect input.
- Humans assume programs can do things that have worked in earlier releases.
- Humans get bored quickly, and skip tests.
- Humans forget tests.
- Humans are expensive.

Regression testing requires the same testing to be performed many times. For example, after fixing a bug, a program should be tested without regard to the particular change. Although a modified statement is an obvious place to look for new bugs, subtle bugs can manifest themselves in distant pieces of software. The likelihood of such bugs is low compared to more blatant problems such as incorrect algorithms. Hence, they get short shrift from programmers during testing.

The UNIX tool-building paradigm encourages designing programs that can be used interactively as well as non-interactively. Such programs can be embedded in *pipelines*. Pipelines are sets of programs, where each program produces output that becomes input for the next program in the pipeline. (The first program in a pipeline does not dynamically consume output of another program, but may for example, read a disk. Similarly, the last program does not produce output that is immediately consumed by another process, but may for example, write to a disk or display.) This is the environment in which Ritchie's remark arose.

In practice, there are forms of input that production programs do not generate. For example, programs do not make typing errors and therefore do not (press the backspace or delete key to) delete characters just produced. Similarly, programs do not enter control characters, such as might be used to interrupt a process. This suggests that Ritchie was too optimistic – even computer generated inputs still test only a subset of a program's interface.

Another problem is that some programs are *never* used non-interactively. For example, the UNIX **passwd** program [2] is designed only to be run interactively. **passwd** ignores I/O redirection and cannot be embedded in a pipeline so that input comes from another program or file. It insists on performing all I/O directly with a real user. **passwd** was designed this way for security reasons, but the result is that there is no way to test **passwd** non-interactively. It is ironic that a program so critical to system security has no way of being reliably tested.

Some programs can be run interactively or non-interactively, but detect the difference and modify their behavior accordingly. For example, virtually all programs that prompt when running interactively disable prompting when running non-interactively. Unfortunately, this makes it difficult to automatically test their interactive behavior non-interactively.

Command languages, such as the UNIX shell, offer no way of dealing with programs that ''know'' they are interacting with a real user. While such languages are rich in control and data structures and can interact with users (prompting and reading responses), they cannot do the same from programs. Command languages in other popular environments

such as VMS and DOS are similarly lacking.

Expect – A Tool for Regression Testing Interactions

Expect [3] is a program specifically designed to interact with interactive programs. Expect communicates with processes by interposing itself between them and acting as an intelligent communications switch. Pseudo-ttys [2] are used so that processes believe they are talking to a real user.

This is useful for regression testing interactive programs. Expect reads a script that resembles the dialogue itself. By following the script, Expect knows what can be expected from a program and what the correct responses should be. The script can specify responses by patterns, and can take different actions on different patterns.

Scripts are written in a high-level language (Tcl – Tool Control Language [4][5]) and support:

- **send/expect** sequences – **expect** patterns can include regular expressions.
- high-level language – Control flow (**if/then/else**, **while**, etc.) allows different actions on different inputs, along with procedure definition, built-in expression evaluation, and execution of arbitrary UNIX programs.
- job control – Multiple programs can be controlled at the same time.
- user interaction – Control can be passed from scripted to interactive mode and vice versa at any time. The user can also be treated as an I/O source/sink.

Expect is actually capable of general use in automating or partially automating interactive programs, however this paper will focus on its use in testing.

I will not discuss a high-level test harness. This can be provided by any number of extant packages or shell scripts that are already in use for testing non-interactive programs. This paper focuses on the low-level problems with program interaction itself which differ significantly from non-interactive testing.

Examples and Guidelines

This section of the paper presents guidelines and examples using Expect to test common interactive UNIX tools, building upon earlier work [7]. Familiarity with the rudiments of Expect and UNIX is assumed.

Example – passwd

The UNIX **passwd** program takes a username as an argument, and interactively prompts for a password. The Expect script in Listing 1 takes a username and a password as arguments, and can be run non-interactively.

```
set password [lindex $argv 2]
spawn passwd [lindex $argv 1]
expect "password:"
send "$password\r"
expect "password:"
send "$password\r"
expect eof
```

Listing 1: Non-interactive **passwd** script. First argument is username. Second argument is new password.

In the first line of the script, the variable **password** is set to the value of the expression in brackets. This expression returns the second argument of the script by using the **lindex** command (**l**ist **index**). The first argument of **lindex** is a list, from which it retrieves the element corresponding to the position of the second argument. **argv** refers to the arguments of the script, in the same style as the C language **argv**.

The next line starts the **passwd** program, with the username passed as an argument.

In the third line, **expect** looks for the pattern ‘‘**password:**’’. There is no action specified, so the **expect** just waits until the pattern is found before continuing.

After receiving the prompt, the next line sends a password to the current process. The \r indicates a carriage-return. (All the ‘‘usual’’ C conventions are supported.) There are two **send/expect** sequences because **passwd** asks the password to be typed twice as a spelling verification. There is no point to this

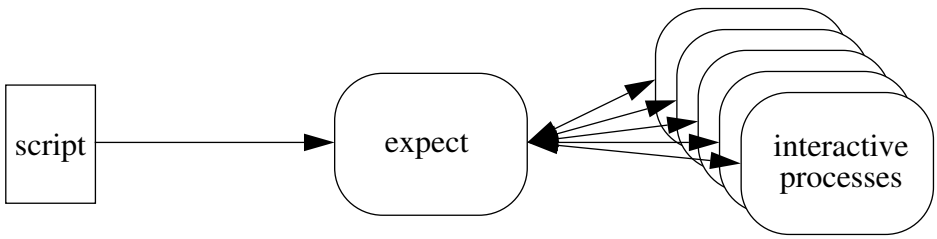

Figure 1: An instance of Expect, communicating with 5 interactive processes directed by a script.

in a non-interactive **passwd**, but the script has to do this because **passwd** assumes it is interacting with a human that does not type consistently.

The final **expect eof** searches for an end-of-file in the output of **passwd** and demonstrates the use of *keyword patterns*. Another one is **timeout**, used to denote the failure of any pattern to match in a given amount of time. Here, **eof** is necessary only because **passwd** is carefully written to check that all of its I/O succeeds, including the final newline produced after the password has been entered a second time.

This script is sufficient to show the basic interaction of the **passwd** command. A more complete script would verify other behaviors. For example, the script in Listing 2 checks several other aspects of the **passwd** program. Complete prompts are checked. Correct handling of garbage input is checked. Process death, unusually slow response, or any other unexpected behavior is also trapped. (The non-interactive functionality of the command is not tested by this script – it is a straightforward task in any language.)

This script exits with a numerical indication of what happened. In this case, 0 indicates **passwd** ran normally, 1 that the user name was bogus, etc. 1X indicates it died unexpectedly and 2X that it locked up, where X is the particular question in **passwd** being checked. Exit numbers are used for simplicity here – descriptive strings could as easily be returned, including messages from the **spawn**ed program itself. In fact, it is typical to save the entire interaction to a file, deleting it only if the command under test behaves as expected. Otherwise the log is available for further examination.

This **passwd** testing script is designed to be driven by another script. This second script reads a file of arguments and expected results. For each set, it calls the first script and then compares the results to the expected results. (Since this task is non-interactive, a regular shell can be used to interpret this second script. As well, it can also be used to test the non-interactive functionality for which **passwd** is responsible, such as checking **/etc/passwd** was correctly updated.) Listing 3 shows a sample data file for testing **passwd**.

```
expect_after {
        eof                                     {exit [expr 10+$question]}
        timeout                                 {exit [expr 20+$question]}
}

set question 0
proc test {args} {
        uplevel {incr question}
        eval [concat expect $args]
}

spawn passwd [lindex $argv 1]
test {
        "No such user"                          {exit 1}
        "New password:"
}
send "[lindex $argv 2]\r"
test {
        "Password too long"                     {exit 2}
        "Password too short"                    {exit 3}
        "Retype new password:"
}
send "[lindex $argv 3]\r"
test {
        "Mismatch - password unchanged"         {exit 4}
        "^\r\n$"
}
test {
        "*"                                     {exit 5}
        eof
}
```

Listing 2: Non-interactive **passwd** script with various tests for behavior at boundary conditions.

The first field names the regression script to be run. The second field is the username. The third and fourth fields are the passwords to be entered when prompted. The last field is the exit value that should match the result of the Expect script. The hyphen is just a placeholder for values that will never be read. In the first test, ''`bogus`'' is a user name that is invalid, to which **passwd** will respond ''`No such user`''. Expect will exit the script with a value of 3, which also appears as the last element in the first line of the regression suite data file. In the last test, a control-C is actually sent to the program to see if it aborts gracefully.

In this example, script arguments are sent to programs literally. However, arguments may also be used to name files or otherwise direct scripts.

For example, the following command sends the contents of the file foo to an interactive process.

```
send "[exec cat foo]"
```

The command works as follows. **exec** executes its arguments as an operating system command. On a UNIX system, **cat foo** returns the contents of the file named foo. Unlike **spawn**, **exec** waits for the command to complete and returns the output, which becomes the arguments to **send**, which sends it arguments to the input of the current process.

Example – suspending sleep

The previous script showed an example of sending control characters to a process, which in response simply exited. Some programs actually use control characters as a normal form of input. For example, UNIX shells typically provide a variety of interpretations for control characters such as control-C (kill foreground process), control-Z (suspend foreground process), control-S (stop output), control-D (input end-of-file), control-O (flush output) and others. A shell script containing such characters will not have the desired effect. Indeed, it does not make sense for a shell script to, say, flush output. If this was intended, the script should not have been written to produce the output in the first place.

Most of these control characters are actually first handled by the terminal driver, which then generates a signal handled by special code in the shell. Since no terminal driver is used when a shell script is executed, it is not possible for the script to call this special code upon encountering these control characters. In fact, shell implementors routinely disable all interactive processing as a matter of course. For example, the shell history functions (which enables the user to recall previous commands) are disabled when the shell is running non-interactively. Again, there is no reason for a shell script to ever need this.

The shell is characteristic, therefore, of a class of programs which function differently when run interactively as opposed to non-interactively. There is no way to verify these interactive elements using shell programming.

The only recourse is to take an approach like Expect, which essentially deceives the shell into running as if it were really interactive. An Expect script can send control characters, history commands and any other commands. The script can also manipulate the environment from underneath, for example, by removing the shell's current directory, or killing a child process of the shell, to check its

```
passwd.exp  bogus     -              -          1
passwd.exp  fred      abledabl       abledabl   0
passwd.exp  fred      abcdefghijklm  -          3
passwd.exp  fred      abc            -          2
passwd.exp  fred      foobar         bar        4
passwd.exp  fred      ^C             -          11
```

Listing 3: Example data file for testing **passwd**.

```
spawn csh               ;# this is a comment
expect "$prompt"        ;# assume prompt is set already
send "sleep 10\r"       ;# run sleep command for 10 secs
exec sleep 1            ;# give time to let sleep begin
send "\cZ"              ;# suspend it
exec sleep 10           ;# wait for 10 seconds
send "fg\r"             ;# let sleep resume
set timeout 5           ;# timeout expect after 5 secs
expect  "*$prompt"      {print "control-Z stopped sleep's clock\n"}
        timeout {print "control-Z didn't stop sleep's clock\n"}
```

Listing 4: Test whether **sleep** counts time while suspended.

response. Listing 4 shows a script which tests whether suspending a **sleep** command actually stops **sleep**'s internal clock.

The script works as follows. A **sleep** is issued for 10 seconds, but is suspended after 1 second. The Expect script than sleeps for 10 seconds, itself, after which it resumes the suspended **sleep**. If Expect then reads a shell prompt, the **sleep** has returned which can only happen if the clock internal to the **sleep** command was still running while it was suspended. If the **sleep** time was indeed suspended, the final **expect** will timeout, since **sleep** will still be running for nine more seconds. (If you run this on most UNIX systems you will find that control-Z does not stop **sleep**'s clock, a counter-intuitive result to most people, but something which must be addressed by implementors and standard writers.)

Example – terminal driver

Scripts can change the default flow of control so that it is not straight-line. Expect supports procedures and the ''usual'' procedural statements such as **if/then**, **while**, etc. A common use of this is to establish limits during conformance testing. For example, one can write scripts to determine the longest variable name supported in the shell, maximum number of arguments to commands in **ftp**, maximum numbers of messages in **mail** message lists, etc. Using shell scripts to solve this, while possible with some programs, requires the process to be restarted for each test. This can be very expensive for limits that are large. In fact, all of the examples listed here are in the thousands.

An Expect script could generate new tests dynamically using a single process. The overhead in such test generation is extremely low by comparison with multiple process creations.

Listing 5 shows a script that determines the longest input line acceptable to the UNIX terminal driver using the Berkeley line discipline in canonical (i.e., line-oriented) mode.

The script works by writing the letter 'a' in a loop, each time testing that it has been echoed properly. When the buffer fills up, the terminal driver echoes control-G's instead of the typed letter. (On a Sun 4 running SunOS4.1.2, this script reported that the terminal driver only accepted 256 characters, a surprisingly small number.)

```
spawn csh
expect $prompt
for {set i 0} {1} {incr i} {
        send "a"
        expect  {
                "\cG"           break
                timeout         break
                "a"
        }
}
print "driver accepted $i chars\n"
```

Listing 5: Determine longest input line acceptable to terminal driver while in canonical mode.

Example – testing buggy programs

The previous examples were completely automated. However, Expect also accepts input from a real user. It does this in two ways. **send** and **expect** can perform I/O with a real user. In fact, **send** and **expect** can perform I/O with any process that has been **spawn**ed, and the user is just treated as another such process for consistency. A very elegant duality appears here – Expect is a process that plays the part of a user, within which, the user can play the part of a process. The user as process is illustrated by the homunculus in the lower-right corner of Figure 2.

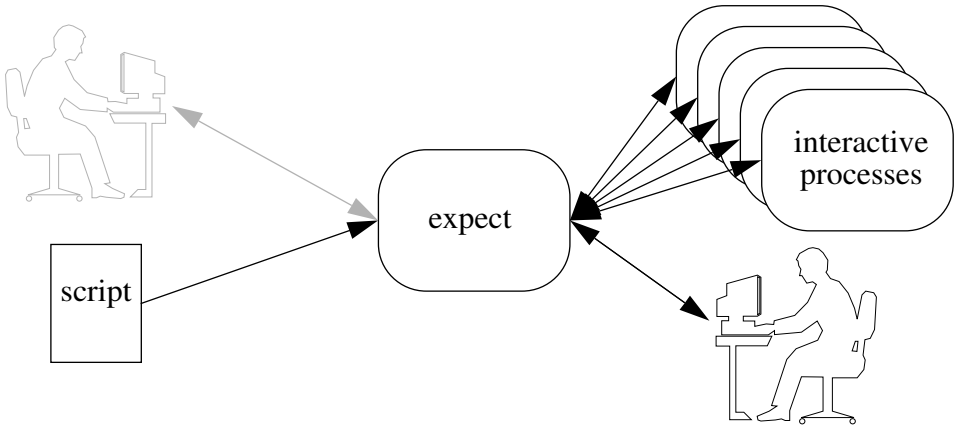

Figure 2: Expect is communicating with 5 processes simultaneously. The script is in control. The user (lower right-hand corner) only sees what the script says to send and is essentially treated as just another process.

The user can take over control from the script and vice versa.

Upon executing the **interact** command, Expect stops reading from the script and creates a direct link between the real user and the process. Thus, it appears to the user as if the process was running interactively in the ''usual'' way. This is especially convenient when testing a program that takes a large number of interactions before reaching a critical part of the program that is buggy and with which the programmer wants to experiment by hand. Listing 6 shows an invocation of an unnamed application followed by some initialization. In a loop, some interactions occur from a procedure named **punish** (to suggest a difficult set of interactions for the application). Control is then passed to the user, who can now directly interact with the application in an attempt to investigate. This is illustrated by the homunculus in the upper-left hand corner of Figure 2.

When the user presses 'X' (or whatever other escape key is chosen), the user begins speaking directly to the Expect interpreter. The user may enter an Expect command such as **return** (return control to the script), **exit** (exit the script), any valid procedure name, or any valid Tcl command, including even another Expect command or procedure definition. This capability is a great convenience in interactive programs that fail only after a large number of interactions. The user may also run the debugger under Expect, essentially providing the user with a programmable debugger. (Very few debuggers include a general-purpose programming interface that can be applied in this way to interactive programs.)

```
spawn ...
initialize
for {} {1} {} {
        punish          ;# punishing procedure defined elsewhere
        interact X      ;# pass control to user
}
```

Listing 6: Run application through a set of punishing interactions, then let user interact. Repeat indefinitely.

```
spawn csh; set csh1 spawn_id
spawn csh; set csh2 spawn_id
send -i $csh1 "send tty\r"
expect -i $csh1 -re "(/.*)\r"
send -i $csh2 "send write $env(USER) $expect(1,string)\r"
expect -i $csh1 -re "Message from .*"
```

Listing 7: Beginning of a script to start two processes that interact with each other – in this case, via **write**.

```
set csh    [spawn csh]
set cshnew [spawn csh.new]
while {-1!=[gets stdin input]} {
        send -i $csh    $input
        send -i $cshnew $input

        expect -i $csh -re ".+\r\n"
        set output $expect_out(buffer)
        expect -i $csh  $output
        if ![string match output $expect_out(buffer)] {
                send_user "detected discrepancy on input $input\n"
                send_user "original csh output $output\n"
                send_user "new csh output $expect_out(buffer)\n"
                interact
        }
}
```

Listing 8: Run two shells simultaneously from the same input, stopping when there is a difference in their output.

Example – Testing interaction between multiple processes

The previous example alluded to the ability of Expect to control multiple processes. Naturally, this is very important when testing interactions *between* processes.

For example, it might be useful to test the response of a running program to various signals from another process. Expect doesn't need to interactively run programs to generate signals, since it can directly call upon any UNIX command (**kill**, in this case, which is non-interactive). However, something like **write** *does* require two interactive processes to test. Listing 7 displays the beginning of such a script. This script starts two C shells. They may be referred to by their *spawn-id*'s, which are temporarily found in the variable **spawn_id**, set as a side-effect of the **spawn** command. (**spawn**'s return value is the UNIX process id.)

Further commands reference the **spawn_id** by the ''**–i**'' flag. In this script, shell 1 executes the **tty** command. The result is used by shell 2 when starting a **write** process directed at shell 1. In this script, both processes are run with the same user id, but it is possible to use multiple logins by spawning **login** first.

Notice that this script uses ''**-re**'' to introduce **egrep**-style regular expressions. While Expect supports both **egrep** and **glob**-style expressions, the **egrep** expressions are much more powerful, and allow very easy access to substrings in matches.

Another use of this multiprocessing ability is to test a new and old program simultaneously until a discrepancy occurs. This is demonstrated in listing 8.

This script reads input from a data file and feeds it to two processes until a difference is found in their output. A more flexible alternative allows the user to drive both programs simultaneously, useful when a user may have difficulty describing a scenario unless actually using and interacting with the program for some time.

Like the UNIX **script** command which records a session, Expect allows interaction to be logged but more flexibly. Listing 9 shows an example.

This script starts two different versions of the same program. In a loop, it listens for output from the programs or the user simultaneously. (The string ''**-re .+**'' denotes a regular expression of one or more characters.) If the user types, the same keystrokes are sent to both processes. If the programs produce output, it is compared, and if there is a difference, an error message is produced.

In this script, one program's output is arbitrarily selected to copy back to the user. Since the other program's output is just a duplicate, there is no point in copying it also. Similarly, one program's output is copied to a log file. An additional

```
set prog    [spawn prog]
set prognew [spawn prognew]

log_user 0                      ;# turn off default logging
set log [open logfile w]        ;# and log to file explicitly

while {1} {
        expect {
                -i $user_spawn_id -re .+ {
                        send -i $prog    $expect_out(buffer)
                        send -i $prognew $expect_out(buffer)
                        continue -expect
                }
                -i $prog    -re .+ {
                        getoutput prog
                        send_user     $expect_out(buffer)
                        puts $logfile $expect_out(buffer)
                }
                -i $prognew -re .+ {
                        getoutput prognew
                }
        }
        if [mismatch prog prognew] report_error
}
```

Listing 9: Run two programs interactively, let user keystrokes go to both until there is a difference in their output. To avoid confusion, only one program's output is returned to the user.

statement could be added to log user keystrokes, although that is usually not necessary since most programs echo them.

getoutput and **mismatch** are not shown here. **getoutput** simply appends the output to a buffer. The **mismatch** procedure is a tiny bit trickier. It has to account for the fact that programs may produce output at different speeds, perhaps due to kernel scheduling slop. So mismatch just matches to the shorter length of either process's output to the current point, saving anything left over for the next time around.

The technique described here is not limited to two processes. Additional processes may be added, each using one more case in the **expect** statement. **mismatch** itself is designed to take an arbitrary number of arguments.

The script itself can remain the same for varying numbers of processes because Tcl can construct new statements at runtime. In particular, **eval** takes an arbitrary list and executes it as a statement. Thus, a list with the appropriate number of cases can be constructed and evaluated on the fly.

Reality and Guidelines

Using the techniques described in this paper, people have written numerous regression and conformance tests for many interactive programs, such as those of IEEE POSIX 1003.2a. The results have been quite satisfying.

Writing such scripts takes experience, just like any programming task. Generally, however, the hardest part is getting a clear specification of the user interface (UI). The facts of life are, unfortunately, that UIs are notoriously underspecified and nonstandard. However, once a specification is available, translation to an Expect script is straightforward.

To date, only a handful of the simplest interactive commands have had UIs specified by POSIX (much simpler and more boring than the examples here). Test assertions are fairly informal in describing what is permitted, with the understanding that a human will actually be dealing with a program and ''understand'', for example, what a ''prompt'' is.

On the other hand, users *are* automating interactive programs, and explicit UI specifications would help. **ftp** is a good (and bad) example. Each message to the user is preceeded by a number, the idea being that a program can read the message number and discard the remainder of the text line which is meant for a human. In practice, a program has to look at both numbers and the messages themselves. The numbers were never clearly enough specified and each implementation assigns different numbers to differing conditions. Nonetheless, the intent was there and **ftp** has been successfully automated.

Designers of interactive programs should account for the possibility of their programs being automated no matter how hard it is for them to imagine. Some programs say: ''*here is a flag to use when running the program via a script. The flag will change (i.e., simplify) the behavior of the program.*'' This is *not* helpful for testing.

Designers of test assertions should be as detailed as possible. Do not assume that interactive programs will only be run by humans. Even screen-oriented programs such as **emacs** and **vi** can and have been automated.

Users who customize prompts should provide a means for programs like Expect to be able to detect this. For example, a generic shell prompt can be detected by the pattern ''`(%|$|#) `''. In practice, few people leave their prompts unadulterated, and Expect users are encouraged to define a prompt pattern for themselves. For most programs, this is conveniently done in the same initialization file at the same time as the prompt itself is defined. For example, a shell prompt and pattern could be defined in a **.login** file as:

```
set prompt="Yes master (\!%)> "
set prompt_pattern="Yes master (.*)> $"
```

Prompt patterns can be outwitted by similar text in normal program output. This is particularly problematic in a login where a message-of-the-day may contain virtually anything including program examples. A '$' at the end of a pattern (shown above) is helpful, as it allows a match only if nothing more follows.

Performance

Performance is essentially the same as has already been described [3], i.e., excellent. Expect is always faster and more reliable than the alternative – a human. Programs which can be broken by sending control-C or other actions sufficiently fast or oddly timed, can be systematically tested by Expect with different inputs and timing until they break.

As described in [7], Expect recently incorporated a mechanism to slow it down to human-like speeds for more authentic testing. Other parameters are available to control human-like variability characteristics in keyboarding.

Current and Future Work

Expect does not provide explicit support for character-based graphics. In particular, the current implementation understands I/O as strictly stream-oriented. Character-based graphics can be manipulated this way, but the script-writer must be aware of issues such as how graphics are written to the display. Although sufficiently expert coding can simulate this (and indeed, a script exists to play the screen-oriented game of **hunt**), several researchers have experimented with the ability to do screen-

oriented interactions. Ideally, a curses inverse is needed to simulate any type of terminal. Researchers are also experimenting with interfaces for describing X or other window system events. These may appear in future releases of Expect.

Applicability

Expect is useful for testing and debugging interactive software. Expect can also be used for building conformance tests of interactive software, such as IEEE POSIX 1003.2a. This paper has presented examples of each of these.

Expect has other uses than program testing. Chief among them is the automation of interactive programs. Nonetheless, Expect has been distributed to over 4000 sites (by request), and the particular use of Expect described herein has proven very popular. Expect has been used to test a wide array of interactive programs, including **tip**, **csh**, many local applications (including Expect itself), and even some non-UNIX applications. While Expect is a UNIX program, it can interact with non-UNIX processes by remotely logging in (e.g., **telnet**, **kermit**) to non-UNIX computers. The language used by Expect does not favor UNIX over any other operating system but is neutral in this regard.

Conclusion

Command shells of UNIX and other common operating systems are incapable of controlling interactive processes. In the past, testing interactive software required a human to press keys and watch for correct responses. After a few iterations, this became quite tiresome. Naturally, people were much less likely to run thorough regression tests after making small changes that they thought *probably* didn't affect other parts of a program.

Expect automates interaction, obviating the need for human effort in regression testing and conformance testing. Using Expect, one can develop automated test suites to assure reliability and consistency with earlier software versions, or conformance with standards, such as POSIX 1003.2a. Expect is also useful for programs that are not yet complete but need interactions in order to evoke failure.

Acknowledgments

This work was supported by the National Institute of Standards and Technology (NIST) Automated Manufacturing Research Facility (AMRF). The AMRF is funded by both NIST and the Navy Manufacturing Technology Program.

Steve Ray, Walter Rowe, Sandy Ressler, Chuck Dinkel, Sheila Frankel, Brian Woodson, and Susan Mulroney provided me with helpful criticism and proofreading of this paper.

I'd like to thank everyone who has recently given me ideas, bug reports and fixes, and porting help, all of which have significantly improved Expect beyond my original ideas and implementation of it. These include John Conti, Steve Summit, Mark Diekhans, Marty Olevitch, Scott Hess, Achille Petrilli, Carl Witty, Stefan Farestam, Jay Shmidgall, John Sellens, Jeff Okamoto, Bob Proulx, Hal Peterson, Wally Strzelec, Ted Gibson, Parag Patel, James Davis, Pete Siemsen, Matthew Freedman, Michael Grant, Phil Shepard, Newson Beebe, Ed Klein, Martin Leisner, Dave Schmitt, Ron Young, Ken Mandelberg, Dongchul Lim, Peter Funk, Karl Lehenbauer, Oliver Kretzschmar, Ian Johnstone, Dave Coombs and, of course, John Ousterhout.

Availability

Since the design and implementation of Expect was paid for by the U.S. government, it is in the public domain. However, the author and NIST would like credit if this program, documentation or portions of them are used. Expect may be **ftp**'d as **pub/expect/expect.shar.Z** from **ftp.cme.nist.gov.** Expect will be mailed to you, if you send the mail message (no subject) **send pub/expect/expect.shar.Z** to **library@durer.cme.nist.gov**.

Disclaimer

Certain commercial products are identified in this article in order to adequately describe projects at NIST. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology.

References

[1] Dennis Ritchie, ''The Evolution of the UNIX Time-Sharing System'', *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Pt. 2, p. 1577, October 1984.

[2] AT&T, *UNIX Programmer's Manual*, Section 8.

[3] Don Libes, ''Expect: Curing Those Uncontrollable Fits of Interaction'', Froceedings of the Summer 1990 USENIX Conference, Anaheim, CA, June 10-15, 1990.

[4] John Ousterhout, ''Tcl: An Embeddable Command Language'', Froceedings of the Winter 1990 USENIX Conference, Washington, D.C., January 22-26, 1990.

[5] John Ousterhout, ''*tcl(3) – overview of tool command language facilities*'', unpublished manual page, University of California at Berkeley, January 1990.

[6] Don Libes, ''*The Expect User Manual – programmatic dialogue with interactive programs*'', to appear as a NIST IR, National Institute of Standards and Technology, 1992.

[7] Don Libes, ‘‘Expect: Scripts for Controlling Interactive Processes’’, *Computing Systems*, Vol. 4, No. 2, University of California Press Journals, November 1991.

Author Information

Don Libes is the author of ‘‘*Obfuscated C and Other Mysteries*’’ and co-author of ‘‘*Life With UNIX*’’. In real life, Don is a computer scientist at NIST where his research deals with manufacturing automation. Don hopes one day to automate himself out of a job. This paper describes the first step. Reach him via U.S. Mail at National Institute of Standards and Technology, Bldg 220, Rm A-127, Gaithersburg, MD 20899. His electronic mail address is libes@cme.nist.gov.