How to Avoid Learning Expect
— or —
Automating Automating Interactive Programs

Don Libes
National Institute of Standards and Technology

Abstract:

Expect is a tool for automating interactive programs. Expect is controlled by writing Tcl scripts, traditionally a manual process. This paper describes Autoexpect – a tool that generates Expect scripts automatically by watching actual interactions and then writing the appropriate script. Using Autoexpect, it is possible to create Expect scripts without writing any actual Expect statements and without any knowledge of Expect.

Keywords: Autoexpect; Expect; interaction automation; Tcl

Introduction

Autoexpect is a tool that generates Expect scripts automatically by watching actual interactions and then writing the appropriate script. Using Autoexpect, it is possible to create Expect scripts without writing any actual Expect statements and without any knowledge of Expect. While this may sound useful only to beginners, even Expect experts now turn to Autoexpect because it is so effective at what it does.

Background

Expect is a tool for automating interactive programs. It is possible to make very sophisticated Expect scripts. For example, different patterns can be expected simultaneously either from one or many processes, with different actions in each case. Traditional control structures such as if/then/else, procedures, and recursion are available. A thorough description of Expect is found in [Libes95].

Expect's language facilities are provided by Tcl, a very traditional scripting language. (A thorough description of Tcl is found in [Ouster].) Traditionally, users write Expect scripts by studying the interaction to be automated and writing the corresponding Expect commands to perform the interaction.

Autoexpect is a program which watches a user interacting with another program and creates an Expect script that reproduces the interactions. In its simplest use, Autoexpect is quite straightforward. For example, consider an ftp session to ftp.uu.net. Normally, it would start out this way:

```
% ftp ftp.uu.net
```

At this point, the user would then interact with ftp. To have Autoexpect automate this, the only difference would be to start the interaction with this line:

```
% autoexpect ftp ftp.uu.net
```

The remainder of the interaction would be the same – the user would interact with ftp as before. Upon exiting ftp, Autoexpect would also exit and present them with an Expect script, by default, called "script.exp". This is graphically shown in the following figure. This mimics the style of the UNIX script command which similarly transparently watches a session and at the end provides a log of the session to the user.

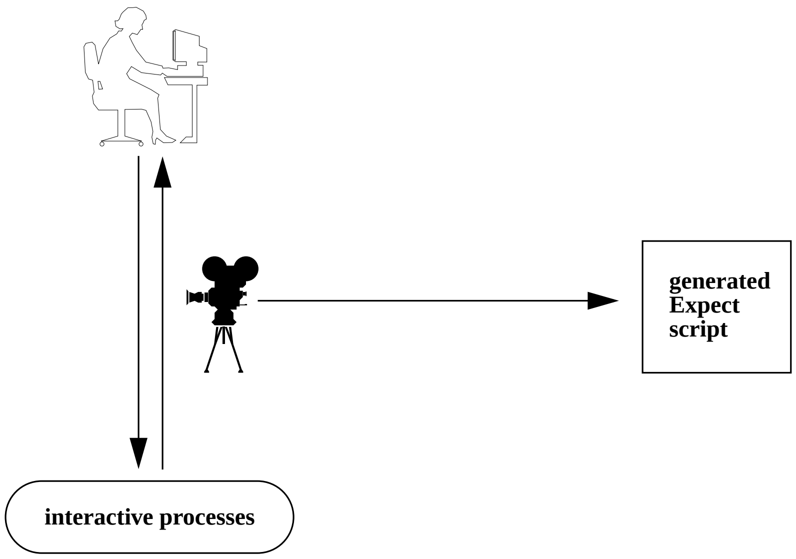

Figure 1: Autoexpect watches an interactive session and generates an Expect script to reproduce it.

Autoexpect can automate multiple interactive applications as easily as a single interactive application. Autoexpect does this exactly the same way that a user does – through the shell. By initially spawning a shell (the default if Autoexpect is called with no arguments), the user can interact, running multiple programs, switching attention from one to another using job control, and other arbitrary interactions. Autoexpect will dutifully note everything and reproduce it faithfully. Job control is supported with no extra effort. For example, the user may press ^Z to suspend a job or ^C to generate an interrupt signal. Autoexpect will generate the appropriate statements to reproduce these so that the interaction can be repeated.

Autoexpect Audience

Autoexpect can be used with zero knowledge of Expect. Autoexpect creates a complete ready-to-run program. Thus, Autoexpect is a good fit for people who know nothing about Expect.

Autoexpect is also useful for users who already have some Expect knowledge. For example, users may want to generalize the resulting scripts, such as by changing repeated sets of statements into loops. Even for Expect experts, Autoexpect is a convenient tool for automating the more mindless parts of an interaction. It is much easier to cut/paste hunks of Autoexpect scripts together than to write them from scratch.

Autoexpect provides a similar value to beginning and intermediate Expect users. An additional benefit is that Autoexpect always provides perfect patterns to match output. Budding Expect users find it useful to run Autoexpect and examine the patterns it has chosen to match program output.

It does not take long to acquire a moderate competency of Expect. Nonetheless, both intermediate and expert Expect users often find themselves in a common scenario: As they are typing something for the second or third time, they start thinking:

> *"Gee, this could be automated using Expect. But that could take 5 minutes to write and 10 minutes to debug and I can simply do the interaction itself in 2 minutes, so I can't justify stopping and automating it with Expect."*

This is sound reasoning – today. Of course, if the user does the same thing tomorrow and again and again, they soon start to lose time by not automating. Only by taking a step back ("*Is this interaction likely to be repeated in the future?*") can the correct choice be made – whether to go ahead and manually interact or to stop and invest the time in automating the interaction.

The shorter and simpler the interaction is, the more likely it is for users to fall into the trap of not considering the advantages of an Expect script. This may seem counterintuitive. In fact, users recognize the value of Expect for long interactions. The more lengthy, painful, or boring the interaction, the more quickly people turn to Expect. But even expert Expect users repeatedly perform interactions that are quick and brief, believing that they're in a rush and that they can't afford to stop and automate a problem that they don't have to think about today.

In short, Autoexpect is worth knowing wherever a user is on the Expect spectrum – from expert to total newcomer.

Potential Pitfalls

It is important to understand that Autoexpect does not guarantee a working script because it necessarily has to guess about certain things – and occasionally it guesses wrong. However, it is usually very easy to identify and fix these problems. The typical problems are:

Timing

By default, Autoexpect produces an interaction designed to run as quickly as possible. However, a surprisingly large number of programs (e.g., rn, ksh, zsh, telnet) and devices (e.g., modems) ignore keystrokes that arrive "too quickly" after prompts. If a generated script hangs while waiting for a prompt at one spot, a brief pause may be necessary before the previous command is sent.

Fortunately, these spots are rare. For example, telnet ignores characters only after entering its escape sequence. Characters are ignored immediately after connecting to some modems for the first time. A few programs exhibit this behavior all the time but typically have a switch to disable it. For example, rn's –T flag disables this behavior (which rn refers to as *typeahead*). The next listing shows how a script must be written to deal with poorly-designed modems.

```
spawn tip modem
expect "Connected"   ;# tip says it has allocated modem
sleep 0.1            ;# pause allows modem to enable UART
send "ATD1234567\r"  ;# otherwise this would be ignored
expect "CONNECT"     ;# and this would hang forever
```

Autoexpect supports a "conservative" mode. By enabling this mode, the generated script will pause briefly before sending each character. The pause is not noticeable to humans but is sufficient to pacify sensitive programs. This mode can be enabled all the time or interactively under control of the user.

It is possible to use precise character inter-arrival times to reproduce the original interaction timings. However, this is not desired by most users who want scripts that interact as fast as possible, not at the same speed as humans. Slowing down scripts to human speeds doesn't guarantee a correct result either since few programs make timing guarantees. Instead, users are at the mercy of the operating system scheduler.

Echoing

Many program echo characters. For example, if a user types "cat" to a shell, what Autoexpect actually sees is:

user typed 'c'
computer typed 'c',
user typed 'a',
computer typed 'a',
user typed 't',
computer typed 't',
user typed \r
computer typed \r\n

A literal rendering into Expect looks like this:

```
send "c"
expect "c"
send "a"
expect "a"
send "t"
expect "t"
send "\r"
expect "\r\n"
```

Without specific knowledge of the program, it is impossible to know if the user is waiting to see each character echoed before typing the next. If Autoexpect sees characters being echoed, it assumes that it can send them all as a group rather than interleaving them the way they originally appeared. Thus, Autoexpect rewrites the fragment above as:

```
send "cat\r"
```

```
    expect "cat\r\n"
```

This makes the script more pleasant to read. However, it could conceivably be incorrect if the user really had to wait to see each character echoed.

The additional \n at the end of the previous expect is added because the terminal driver normally echoes a return character with a return-linefeed sequence. This knowledge is not wired in to Autoexpect. In this situation, Autoexpect merely consumes as many characters as possible until the user resumes typing. This makes scripts very “tight”. Rather than three expect commands (one each for the echoed command, the command output, and the subsequent prompt), only one expect command is generated which matches everything.

As an aside, most human-written scripts would not bother with the “expect "cat\r"” either. After all, Expect itself will just skip over extraneous output. Autoexpect could be made to make this optimization except that it raises the concern that the following output could resemble the command. In that case, the generated script would execute the next send before the desired output had actually appeared.

Change

By default, Autoexpect records every character from the interaction in the script. This is desirable because it gives the user the ability to make judgements about what is important and what can be replaced with a pattern match.

On the other hand, the generated scripts are not going to be correct if interactions involve commands whose output differs from run to run. For example, the “date” command *always* produces different output. So using the date command while running Autoexpect is a sure way to produce a script that will require editing in order for it to work.

Autoexpect supports a “prompt” mode. In this mode, Autoexpect will only look for the last line of program output – which is usually the prompt. This handles the date problem (see above) and most others.

Style? Not!

Autoexpect scripts will not be mistaken for humanly-generated scripts. Autoexpect uses a variety of features that were always intended for machine-generated scripts. For example, all send commands generated by Autoexpect use the “--” flag, as in:

```
    send -- "more"
```

The “--” suppresses any possible interpretation of the next argument as a flag. Obviously, this is redundant here. “more” is not a flag. However, by using the “--”, it is not necessary for Autoexpect to check whether or not “more” is a flag. Thus, for Autoexpect, it is simpler to use the more verbose form. For users, it is simpler to use the less verbose form. There are many operations in Expect that follow this convention. Thus, Autoexpect scripts tend to look somewhat verbose and peculiar.

There are useful things to be gained by studying the output of Autoexpect. For example, it is possible to see how any string must be quoted in order to use it in a Tcl script simply by running Autoexpect and typing the strings in. But in general, Autoexpect should not be held up as a model of human programming style. That is not its goal.

Implementation Notes

Autoexpect is implemented as an Expect script. The Expect script is rather interesting in itself, however discussion of the techniques is beyond the scope of this paper. One relevent item is that Autoexpect includes a hefty template, inserted at the beginning of each script describing things about the generated code – just in case the user *does* look at the script, they should have a little introduction to Autoexpect's surprising style.

Despite its implementation in pure Expect, the script easily keeps pace with interactive use. For instance, large chunks of output are processed with a single read() and appended to an output buffer. Only when the user types a character are the script output buffers flushed. And since users type slowly (relatively speaking), this processing is not noticeable. This is similar to the approach taken in Kibitz, and also in Tk in general in which events are handled entirely using Tcl scripts ([Libes93]).

Todd Richmond (Legato, Inc.) demonstrated the feasibility of automating Expect scripting by modifying Expect itself. Written in 1991, this experiment was never publicly released, in part merely because it was based on a much earlier version of Expect and no one had the time to integrate it back into the current version.

In retrospect, a script-based Autoexpect provides much more flexibility and without loading down the Expect core. Like Tcl and Tk, we should resist changes to the Expect core that can be more profitably accomplished using Tcl itself.

Work In Progress

The publicly-distributed Autoexpect produces straight-line code. This may sound simplistic, but it is valid.

Work is currently in progress on "loop rolling" – the removal of statements by creating loops to produce the equivalent effect. This section describes several example problems that are being studied.

Creation of Counted Loops

Consider the following sequence. It has a repeating send/expect sequence:

```
send A
expect B
send A
expect B
send C
```

This can be rewritten as:

```
for {set i 0} {$i<2} {incr i} {
  send A
  expect B
}
send C
```

This rewrite is not particularly valuable with only two repetitions, but it shows the possibility for handling more repetitions.

Creation of Output-dependent Loops

Consider the following sequence. It has a repeating expect/send sequence:

```
expect B
send A
expect B
send A
expect C
```

If this interaction is repeating until the appearance of C, this is better rewritten as:

```
expect {
  B {
    send A
  }
  C
}
```

If the interaction is not repeating until the appearance of C, this form can still be used although it might be better not to use this form unless there are many repetitions (however "many" is defined). The earlier counted-loop may be more meaningful, however.

Other Problems

These solutions must be generalized. For example, repeated statements may consist of multiple send-expect sequences, not just one. Similarly, nested expect statements must also be supported. For example, consider this interaction:

```
expect B
send C
expect D
send E
expect B
send C
expect D
send E
expect F
```

This could be rewritten:

```
expect {
    B {
        send C
        expect D
        send E
    }
    F
}
```

User interactions occasionally require commands other than send and expect. So these other statements must be handled as well.

Unfortunately, there are no optimal rewriting algorithms. Some script fragments can be rewritten in several ways. Straight-line scripts easily solve the goal of reproduction. The more difficult and less definable goals are appropriateness to a particular task. And what makes good sense for one task may not make much less sense for another. One approach to consider is having a user-driven generator with a graphical interface. The graphical interface would present various ways of "rolling" the code. As the user would select and unselect them, the code would be shown rewritten in the various forms.

Conclusion

For many scripts, Autoexpect saves substantial time over writing scripts by hand. Even Expect experts will find it convenient to use Autoexpect to automate the more mindless parts of interactions. It is much easier to cut/paste hunks of Autoexpect scripts together than to write them from scratch. And beginners may be able to get away with learning nothing more about Expect than how to call Autoexpect.

Acknowledgments and Availability

Thanks to Jeff Bowyer (Institute of Computer Science, Masaryk University, Brno, Czech Republic) and Henry Spencer for assistance in the development of Autoexpect. Thanks to Steve Ray and Jim Fowler for providing suggestions which improved this paper.

Expect and Autoexpect are 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.

References

[Libes93] Libes, D., "Kibitz – Connecting Multiple Interactive Programs Together", Software – Practice & Experience, John Wiley & Sons, West Sussex, England, Vol. 23, No. 5, May 1993.

[Libes95] Libes, D., “Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs”, O’Reilly and Associates, January 1995.

[Ouster] Ousterhout, J. K., “Tcl and the Tk Toolkit”, Addison-Wesley, 1994.

Author Biography

Don Libes is the creator of Expect as well as the author of its definitive text, Exploring Expect (O’Reilly, 1995). Don has written over 80 computer science papers and articles plus two UNIX classics: Life With UNIX (Prentice Hall) and Obfuscated C and Other Mysteries (Wiley). Don is a computer scientist at the National Institute of Standards and Technology. Reach him electronically as libes@nist.gov.