Writing a Tcl Extension in only... 7 Years

Don Libes
National Institute of Standards and Technology
Gaithersburg, MD

Writing a Tcl Extension in Only ~~Three~~ ~~Four~~ ~~Five~~ ~~6~~ 7 Years

Don Libes
National Institute of Standards and Technology
Gaithersburg, MD 20899
libes@nist.gov

Abstract

Expect is a tool for automating interactive applications. Expect was constructed using Tcl, a language library designed to be embedded into applications. This paper describes experiences with Expect and Tcl over a seven year period. These experiences may help other extension designers as well as the Tcl developers or developers of any other extension language see some of the challenges that a single extension had to deal with while evolving at the same time as Tcl. Tcl and Expect users may also use these ‘war stories’ to gain insight into why Expect works and looks the way it does today.

Keywords: Expect; Lessons learned; Software archaeology; Tcl

Expect – What is it?

Expect is a tool for automating interactive applications such as telnet, ftp, passwd, fsck, rlogin, tip, etc. [Libes95] Expect is also capable of testing these same applications. Expect is implemented as an extension to Tcl [Ousterhout]. Using Expect with other extensions is straightforward. For example, with Tk, interactive applications can be wrapped in modern graphic user interfaces allowing a program that was originally command driven to be controlled with buttons, scrollbars, etc.

Expect was first released in 1990 and rapidly became and remains today the standard tool for automating character-oriented applications. It is now used by hundreds of thousands of companies and institutions around the world. It is sold and supported by several vendors and is distributed for free from NIST[1] and with many vendor software distributions. Expect has become the de facto recommended solution for many problems faced by programmers, system administrators, and users. Expect is described in many FAQs, there are chapters on Expect in several books, and Expect is recommended by dozens of authoritative papers and articles.

Experience Papers – Why?

It is useful to look back over lengthy periods of time. This allows observations that are uncommon. Most ‘experience’ papers have only a year or two to draw upon. Expect is in the unusual position that it predates almost all other extant Tcl extensions. This makes it an excellent candidate for study in the hopes of finding lessons that we can apply to more recent or even future extensions.

A second reason is that the Tcl developers are not required to consider the ramifications of their own decisions on extensions. The Tcl developers are in the enviable position of being able to change Tcl itself as needed. Needless to say, building an extension can be an entirely different experience depending upon whether you have the freedom to modify Tcl or not.

In 1995, Phelps described many difficult problems faced and solved in the TkMan and Rosetta projects during a two-year period [Phelps]. Interestingly, that paper is entirely different than this one. For instance, one of the crucial difficulties with TkMan was how to achieve high performance, something that is not an issue for Expect. (“Realistically, blazing speed is hardly needed in a program that simulates users” [Libes91].) In that and other respects, this paper is complementary to Phelps’ and similar papers that offer more traditional lessons learned.

Tcl – Swamp or Savior?

It is worth pondering how Tcl affected the development of Expect. How did Tcl help Expect? Would a different approach have been better? Can these questions be answered in retrospect? After seven years of Tcl, is it possible to be objective about it? These questions are difficult to answer and for the most part, I do not have cogent answers. Although I will address these points indirectly throughout the paper, I encourage readers to try and make up their own minds as I present some of the many war stories in Expect’s history.

Much of Expect’s beauty is due to Tcl. Tcl really does well what it was intended to do – be a glue language. Tcl makes it easy to incorporate new functionality. Expect takes a general approach towards its problem domain. It is usually possible to solve a specific case with a specially-written program (e.g., ncftp) [Dalrymple]. The advantage of Expect is that it applies to any

1. http://expect.nist.gov

interactive program. So once you've learned how to apply Expect to one program (e.g., ftp), it's trivial to apply it to another (i.e., telnet, passwd, crypt).

Much of the flexibility is due to Tcl itself. Tcl provides Expect with the basic infrastructure for variables, procedures, expressions, etc. Expect need only focus on the parts of the task specific to interactions with processes. At least, that's the theory.

Tcl tries very hard not to force a particular view of the world. For example, Tcl allows OO (object-oriented) or non-OO programming – both simultaneously. And Tcl permits the addition of new control structures. In fact, I didn't even realize that's what I was doing until years later when I tried to implement Expect in other languages.[1] These are examples of approaches that Tcl encourages. And Expect follows Tcl's suggestions – except in the following cases:

- Expect got there first (i.e., there was no suggestion until later), and
- Tcl's approach was confusing, painful, or I just didn't understand it.

I will give examples of both of these as well as talk about problems caused by myself and other outside influences. Having introduced key concepts (and sagas), I will then return to cover more examples of difficult interactions with Tcl as well as other insights.

Tcl's Approach – Unpleasant or Inadequate

Some of Tcl's ideas which at first seem to fit well with Expect, did not actually fit well. This section presents several examples.

Null Strings

For many years, Tcl strings did not support nulls. This was an understandable philosophical decision. C strings were naturally null terminated, so using them simplified the C API. And Tcl as a glue language had little need for nulls. Fortunately, Expect didn't have much need either. Users don't see them – they have no printed appearance – so why would a program send nulls? In fact, there are programs with interactive interfaces that do send and receive nulls. For example, curses-based programs send nulls for screen formatting. In 1990, Expect began providing null support. Initially, it was very basic – nulls were stripped out of strings before being matched. Eventually, the support grew more sophisticated and capable, but the user interface was always unnatural due to the lack of Tcl's support.

After years of begging for null support in the core, the Tcl developers have scheduled it for Tcl 8.0, expected to appear sometime in 1997 – seven years after Expect's first workaround.

Scoping & OO

Originally, Tcl's scoping rules were very limited. I found them unpleasant enough that I made Expect's commands internally look in both the local and global scopes for special variables such as spawn_id and timeout. This simplified many scripts but it was not the default Tcl behavior so it had to be explained as a special Expect feature. Had Expect been designed after many other extensions existed, I might have felt less inclined to do this, but there was no tradition being violated at the time, so I didn't think very hard about it.

A related issue is that Expect defaults to using global variables (such as spawn_id) rather than insisting that they be passed as parameters. Again, this simplifies most scripts. The vast majority of users never manipulate more than one interactive process at a time. The result is faintly reminiscent of an OO implementation. At one point, I considered implementing an object-based (OB) version of Expect in the style of Tk. But for most Expect scripts, an OB version would be less practical. Contrast a traditional Expect sequence with the OB style approach:

```
# traditional
expect "Login:"
send "$username\r"

# object-based
$telnet expect "Login:"
$telnet send "$username\r"
```

I'm not aware of any Tcl extensions that actually started with an OB orientation and later dropped it. However, CGI.pm, a Perl module for CGI scripting provides such an example [Stein]. Early releases had an OB view. While it was clean as far as scoping and namespace control, users rarely used multiple CGI objects and the OB interface was cumbersome. Thus, a non-OB/OO interface was created. In practice, CGI.pm scripts are generally a mixture of OB and non-OB/OO programming.

Pattern Strings

When I began using Tcl, I found the rules for pattern creation to be quite confusing – partly because patterns

---

1. I was familiar with this practice having used Lisp for many years. That's probably why I didn't think too much of it at the time I began using Tcl.

are not built into the grammar but are simply represented as strings which the pattern matcher would reinterpret. I exacerbated this problem further by using a list to provide alternation over glob patterns. This caused yet another round of interpretation, so that matching trivial strings such as "No match" had to be written: {No\ match}. Tcl 6 incorporated a regular expression engine, thereby providing alternation for free. This required the rewriting of many patterns but it was worth it, drastically simplifying things.

Tcl's general quoting conventions are still a source of misunderstanding for many. This phenomenon is common enough that people have humorously given it a name: Quoting Hell. Part of the problem is that the conventions are different than anything people are used to – such as those of the shell. Tcl and shell quoting bear a resemblance to each other but it is superficial. Tcl's quoting rules are simpler and more straightforward. For this reason, it is easy to master them – as long as a good explanation is at hand. Unfortunately, because the quoting appears so similar at first, many people prefer not to learn the rules and begin to program using another language's conventions, thereby getting themselves confused and leaving the impression that Tcl's conventions are even more illogical than the shell.

In short, Tcl's quoting hell is actually bliss. The conventions are so simple and regular that they are trivial to learn. By comparison with most other languages, Tcl syntax is a pleasure to use.

Other languages, such as Perl, take the approach that patterns are a special type. This has advantages and disadvantages. One advantage is that you can have rules specifically for pattern formation, making patterns easier to write and more readable once written. The corresponding disadvantage is that additional rules themselves can be a burden on the bulk of users who cannot afford the time or brainpower to master the large number of them [Friedl].

It is worth considering the analogy to expressions. In early releases, Tcl's expr command did not support the traditional functional notation that it does today. For example, sin($x) had to be written [sin $x]. The current expr command is much more like a traditional language, but at the same time unlike anything else in Tcl. I, personally, was not thrilled at the change. Indeed, it is widely agreed that some of the shortcuts in the expr command should be avoided. It is interesting to contrast this with Perl where the number of shortcuts is one of the attractions of the language.

Expect Got There First

In some cases, Expect's design choices were made before Tcl's. When Tcl later changed, this occasionally resulted in outright conflicts that required substantial code rewrites. In other cases, Expect's choices were incorporated by Tcl or accepted by the Tcl community as a standard solution. But in either case, the cost of leading the way was expensive.

Command Name Collisions

The first release of Tcl had no file I/O and therefore no open, close, etc. Although Expect did not do generalized I/O, "close" was a natural name for one of its functions. So when Tcl finally incorporated file I/O with a command by the same name, there was a collision. Fortunately, it was easy to tell which close was intended by a trivial examination of the arguments. In fact, Expect's close command calls Tcl's close command if that's what the arguments indicate. Similar fixes were not possible in other cases. For example, Expect's send command collided with Tk's send command and there was no way to distinguish which was intended from the arguments alone. A general solution was adopted of making all Expect commands that did not already begin with 'exp' available with an 'exp_' prefix.

After running into enough of these collisions, I eventually adopted an extremely conservative policy: Before declaring each Expect command, Expect checks whether a command already exists by that name. If so, the command is only declared with the exp_ extension. Thus, it doesn't bother Expect if other App_Inits redefine Expect's commands either before or after Exp_Init. (It is permissible to redefine commands, although for obvious reasons, this is not common practice among other extensions.) It is interesting to compare Expect to BLT [Howlett]. BLT started using the blt_ prefix on all commands in 1993 but dropped it in 1996. Besides being notationally distasteful, it was difficult to support both the blt:: and blt_ flavor for each command. BLT's author also wanted to demonstrate the need for namespace support in the Tcl core.

A related issue arose with Expect's exp_continue command. exp_continue causes the currently executing expect to restart, much like continue restarts a while loop. Both continue and exp_continue make sense (and do different things) from within an expect action. Their implementation depends upon a shared namespace which include values such as TCL_ERROR and TCL_CONTINUE. Unfortunately, Tcl provides no support for allocating these values uniquely.

Other examples of unmanaged collisions persist as well, such as zombie process identifiers. To be fair, Tcl does manage a large number of resources. Yet it is surprising that a language intended specifically for extension fails to address many areas of obvious conflict [Libes97].

Substitutions

Tk's bind command demonstrates how to obtain information using a substitution mechanism (i.e., %). This idea didn't become known to me until it was too late. By the time I became aware of Tk, Expect already used reserved variables names (e.g., expect_out, spawn_out). The % mechanism is unquestionably easier to read simply because it dramatically shortens commands.

On the other hand, substitution has its drawbacks. For instance, it is confusing when used with other extensions or commands that perform similar substitutions (e.g., format inside of a bind). One of the more flexible extensions in this regard, Oratcl, performs substitutions using @ but lets users override that character dynamically, perhaps with the expectation that yet another extension might come along and choose it too [Poindexter]. In fact, there really aren't that many 'good' characters to introduce substitutions and Tcl provides no mechanism for handling the problem in a formal way.

In any case, the 8.0 compiler may put a damper on further enthusiasm over substitutions since continually regenerated commands make it very difficult to take advantage of code previously generated by the compiler.

Event Management

In version 7.5, Tcl began providing event management, in large part taken from Tk. Before this unification, Expect had to provide its own event management for the case when Tk was not present. For example, the interact command necessarily waits for input from two sources at the same time – the user and the process. The obvious solution uses select but this doesn't work on some systems. So Expect knew how to use poll too. And on systems where neither select nor poll worked on ptys, Expect achieved the same result by using multiple processes, one to wait for each input stream. This was an ugly but standard maneuver in UNIX V7 days for management of multiple streams by any process such as cu and telnet [Nowitz].

Tk's style of event management was much more flexible than that of Expect and required significant changes. For example, consider the following code:

```
expect "password:" {
      send "$password\r"
      exp_continue
}
```

This tells expect to wait for "password:" and when found, send back a password, and to do this repeatedly. This is a typical sequence for handling a passwd-like program where it is not known in advance how many times the password will be prompted for. With Tk-style events, just about anything can happen while expect is waiting. This includes spawned processes shutting down or even being replaced with different ones. In such situations, not only could the action be inappropriate but so could the expect itself. I still see people be surprised by this flexibility.

Possibly one of the reasons this is surprising is that not all of Tcl's file operations were similarly modified to support events. For example, consider these two operations which both read a line from the standard input:

```
gets stdin
expect_user \n
```

Only the expect command allows events to be processed. In contrast, Tcl's gets command blocks all events until the gets command returns.

Needless to say, a lot of work went into Expect in order to support four styles of event management (select, poll, Tk-style, V7). And this had to be rewritten when Tcl introduced the concept of the notifier.

Solving Problems that had Nothing to do with Expect

Ferreting out bugs in Tcl itself is an example of a task that had nothing to do with Expect itself. Less obvious problems grew out of a need for functionality that really had nothing to do with Expect per se. For example, a debugger was de rigeur; however, a suitable one did not exist so I stopped Expect development to write one [Libes93]. A number of pieces of Expect fall into this same category, such as signal handling, time formatting, and others. Some of these are now available elsewhere. For example, Tcl now directly supports the clock command and TclX provides signal handling [Diekhans]. Unfortunately, at the time, I couldn't afford to wait a year or two.

Designing Software Myself (or: What I Can't Blame on Tcl)

Some of the design choices I made were poor. Most have been corrected. In a few cases, I decided it was preferable to entertain the few complaints about Expect's design over the thousands of complaints had I broken everyone's code. One example is the default timeout of expect. While the concept of an implicit timeout was innovative, it is now clear that the timeout should be off by default. Instead, expect times out after 10 seconds. Although this is changed trivially in a script, it nonetheless is something that has always bothered me. Of course, I'm equally bothered by the many other programs that also choose arbitrary time outs such as ping (20 seconds) and rsh (75 seconds). It is ironic that Expect is useful in dealing with so many other programs that have capricious timeouts. Who knows where all these magic numbers come from?

In some instances, I intentionally introduced pitfalls although I tried to do it as gracefully as possible. An example is the interpretation of expect when given a single argument. Consider:

```
expect {foo bar}
```

Expect interprets this as a request to wait for the string "foo bar". But a naive user might have intended this to mean: wait for "foo" and then execute "bar" after writing similar statements with only different formatting:

```
expect {
     foo bar
}
```

Clearly Expect has to make a guess. The heuristic Expect uses is fairly sophisticated although it is hampered by the impossibility of seeing the original quoting. In particular, the following two statements invoke Expect commands with the exact same arguments:

```
expect "foo bar"
expect {foo bar}
```

It is impossible for Expect to know how the statement originally appeared, and thus the user's implicit hint over whether the argument was a simple string or a control structure is lost. A similar problem exists with the interpretation of newline (or for that matter, any formatting character) which is represented the same way whether it was originally used for formatting or specified as an explicit string. Consider:

```
expect "\n"
expect "
"
```

When allowing this double interpretation, I knew that people would occasionally step into it, but I held my breath in hopes that the number of people would be few. And the heuristic is good enough that people have to express things really unnaturally in order to be tripped up. For example, intending "foo bar" as a control structure is extraordinarily unlikely because control structures are only useful when you have multiple patterns. With one pattern, the user isn't even going to bother embedding the action in the expect command. And if they perhaps got to one pattern-action by starting with several and deleting the others, they'll still end up with the traditional formatting (with embedded newlines). Fortunately, history has born out my optimism. In total, only three people have ever reported being tripped up by this. As a final note, the heuristic can always be avoided entirely by calling expect with the -brace or -nobrace flags. In fact, Expect does this internally to avoid recursion.

It is interesting to compare this problem with Tcl's switch command. In fact, Expect was patterned after that (nee case). One particularly unfortunate drawback of bracing an argument list was the loss of evaluation. This makes the behavior of the two different forms quite different. While the braced behavior is a natural outcome of Tcl's normal command evaluation, the lack of substitution and other features renders this form of expect near useless – a high percentage of expect patterns incorporate variable substitutions and special characters – such as "$prompt" and "\n". Thus, Expect does another round of evaluations in the style of the expr command. A modern approach to this would use the subst command but Expect still takes advantage of private Tcl interfaces simply because this work was done well before the introduction of subst (and the interfaces have continued to work).

Portability – Not!

One pleasant aspect of Tcl is the portability that it provides, both to the end-user and the extension writer. Of course, the extension writer must strive for portability as well. And that is not always an easy task. Although the primary aim of this paper is to present experiences directly relevant to the Tcl community, I feel obliged to give just one example of some of the effort that was invested in Expect to address portability in its own problematic areas. Hopefully, this material can be of use to vendors and the standards community – and amusement to others.

Many people think the existence of POSIX has solved all of our portability problems [POSIX]. Alas, it has not. There are two reasons why:

- no POSIX support on pre-POSIX systems
- POSIX doesn't standardize everything

These simplistic observations are more complex than they might at first appear. Part of the problem is that POSIX is not a simple standard. POSIX is really a family of standards – each part of which appeared at a different time. And a particular operating system can pick and choose much of which the vendor wants while still using the term POSIX to describe it. Of course, many of these 'features' can be detected using simple tests during the installation process. Alas, some vendors seem determined to make that as difficult as possible. For instance, one vendor helpfully provided all of the possible POSIX include files but for libraries that didn't exist. In fact, much of what Expect does requires that features work in a certain way interactively, so Expect must successfully compile and run many test programs in order to figure out how a system behaves. And workarounds must be provided for each missing piece or divergent behavior.

Another problem is that many vendors disable POSIX features unless specifically requested, but if POSIX features are requested, then non-POSIX extensions are disabled. And because POSIX does not cover many areas, Expect necessarily must use non-POSIX extensions on all systems. Thus, Expect never requests POSIX support in the official way.

There are a surprisingly large number of things that POSIX does not define. For Expect, one of the biggest problem areas concerns pseudoterminals, or *ptys* for short. Ptys are the operating system abstraction that allow Expect to make a process believe it is interacting with a real user at a real terminal. Unfortunately, there is no standard pty interface.

Expect supports seven major variations of pty. Some systems attempt to address portability by supporting two or even three of the variations. Of course, they are never the same two or three, and systems rarely document which are the preferred interfaces. Most of these variations have subvariations, making the number of pty interfaces over two dozen. Here are some of the pty behaviors with which Expect has to deal:

- How is a pty allocated? On some systems, the file system must be searched for them. On others, a function (e.g., getpty) is provided. There is no standard name or behavior for such functions.
- What accessibility/usability tests have to be applied to the pty? Some systems return a master but you have to successfully open it and the slave before you can really be sure it is valid. Some systems require example I/O be executed to test it as well – just opening isn't enough.
- How is the pty initialized? Are the terminal modes of the pty pre-initialized (and what are they? Can they be changed and on what basis? I.e., system-wide, session-wide? Or must we assume the pty is in some unknown state? (Some systems pre-initialize the pty, but since this isn't documented we can't rely upon it, thus we spend time re-initializing anyway.)
- Is the pty drained on slave-side close after some specified time period. Can the time be changed/disabled? Stream-implementations do this, but with varying time periods.
- Do slave side operations have to be acknowledged? Some implementations require that the master 'approve' of certain slave system calls that affect the pty.
- How does the master-side detect a slave-side close? Some systems have read() return -1 with errno = EIO. Some force the use of select. Some have select return a readable fd while some have select return an exception fd. (None do what I would consider the right (and obvious) thing with read which is return 0 like any other file descriptor.)
- What ioctl's initially have to be applied to the slave side? Some systems want I_PUSH ldterm; others also want ptem and ttcompat. Some don't want anything.
- Is setuid required? Must an entry be made in wtmp/utmp? What is the interface?
- What #includes are necessary for all of this? Each system seems to have its own unique collection of include files to describe this mess.

Obviously ptys are a quagmire. But the complexity doesn't end there. Many of the operations performed on ptys are also nonportable. This has nothing to do with ptys – these same operations are nonportable on ttys, too. For instance, there are a variety of ways to gain a controlling terminal. The usual answer seen on comp.unix.programmer is *setsid() followed by open("/dev/tty",...)*. However, this doesn't actually work on a lot of systems and there's no right or wrong because, again, POSIX leaves this undefined.

Getting the answers to questions like these is often a painful ordeal. While man pages exist for pty(4) and setsid(1), there are no man pages for interactions between them. The man pages that do exist, never provide complete explanations, leaving trial-and-error as the mechanism to finding how things really work. Of course, this is really bad. Undocumented behavior often changes from one release of the same OS to another. (Actually, even *documented* behavior often changes.) So contributed fixes that are ifdef'd for a particular vendor are often inappropriate, always suspect, and there is no trivial way to address the problem. It is depressing to receive patches from people that cannot be used because there is no assurance that it won't break support for other versions of the identical OS or even identical versions but on different platforms.

With ptys, I was forced to address each different pty interface. There was no way of avoiding it. A different solution was taken with regard to setting terminal modes. Terminal modes fall into the category of 'semi-standard'. There are well-known interfaces (two, of course) for setting terminal modes: ioctl and tcget/setattr. The latter is partially specified by POSIX. So Expect makes use of this specification when possible.

Unfortunately, POSIX only defines a subset of terminal modes. Vendors always extend the modes. In order to allow users to make use of native extensions in their familiar tongue, Expect merely calls stty after encountering anything it doesn't recognize. Calling stty is very much in the spirit of Expect which encourages reuse of existing programs whenever possible.

Unfortunately, stty cannot portably be called directly by the user for several reasons:

- Some stty implementations require redirection differently than others. BSD stty traditionally applies to the device on the standard output while SV stty applies to the device on the standard input. The obvious solution is to redirect both. Alas, a feature of modern BSD is to complain if the 'wrong' descriptor is redirected. So if the user invokes Expect's stty command with unrecognized arguments, Expect internally calls stty with the correct redirection.
- Common stty modes are interpreted differently. For instance, even though most users intuitively want to specify 'raw' mode, there is no 'standard' definition of such a mode. Indeed, some sttys don't even have such a mode.
- Window size information is nonstandard. One would think that the concept of 'rows' and 'columns' is not particularly challenging. Yet there is no standard output syntax for window size information. Not surprisingly, internally each system uses different ioctls to support it. Some systems don't support window size at all. Since window size information is so frequently used, Expect handles this itself bypassing stty. This enables scripts to be portable.

Configuration

Tcl was originally envisioned as a small language library for embedding in applications. At the first presentation on Tcl that I attended, I immediately saw the merit in this. And ignored it. Expect reversed the idea, promoting Tcl to be the important application with just a small augment of process-control commands. Viewed this way, it made sense to ignore some of Ousterhout's early recommendations, such as:

- *Include the Tcl distribution with every Tcl-application.* Expect never did this – to its benefit. As Tcl became enormously popular, it would have been silly to reship the bulky Tcl distribution over for each Tcl application. And for the most part, Expect didn't care which version of Tcl it had available. Early on, however, I did receive many complaints that Expect was too hard to use because it required people to install Tcl first.
- *Embed Tcl's version number in the application version number.* This was a reaction to the incompatibilities of different Tcl versions. Since Expect was always able to support the last several versions of Tcl, embedded Tcl's version number seemed pointless. (This may not remain true in the future, though.)

As Expect was ported from one system to another, the configuration difficulties became enormous. There were more and more configuration choices in the Makefile and users were having a hard time figuring out how to configure things. In early 1993, Rob Savoye at Cygnus Support stepped in and automated Expect's configure process using GNU Autoconf. Autoconf had problems of its own, of course, but they were minor by comparison with manual configuration. Although the effort to maintain Autoconf configure scripts is high, the end result is that user's lives are vastly simplified and developers escape the daily barrage of installation questions. Cygnus, of course, benefited too. They had dozens of different machines on which they wanted to support Expect and installing it on each manually was unwieldy and expensive.

Making the switch to Autoconf required a bit of faith to take the plunge, and Tcl dove in as well several months later. This delay didn't directly hurt Expect, but once again it felt like the tail wagging the dog.

In contrast to Autoconf, Tcl caused Expect much worse configuration problems. For instance, Tcl generated a set of compiler flags defining whether certain functions or include files were available. However, these compiler flags were kept private so Expect had to repeat the logic and regenerate the flags during its installation. Only in recent versions, did Tcl finally make available its configuration options in a public way – through tclConfig.sh. Unfortunately, although public, these interfaces remain undocumented.

Problems Successfully Avoided

In contrast to many of the reinvented wheels, Expect also avoided many. Whether this was wisdom or luck is an open question. For instance, early versions of Tcl had no file support. This wasn't a major problem for Expect since it was possible to use existing utilities to read and write files – just as a user would. Indeed, an early paper demonstrated how to retrieve some information out of /etc/printcap by reading it – using ed [Libes 91]! (To be fair, ed did some scanning to locate the information that would have been harder without it or some other similar tool.)

More recent problems revolve around shared/dynamic library support. I steadfastly refused to add shared library support to Expect until the Tcl support had been considerably shaken out – this took several years. Even then, problems remain. For example, Tk's configure does not distinguish between the libraries required for Tk versus those required purely for Tcl. Because some systems object to repeated library specifications when building dependency lists for new shared libraries, Expect has to figure out which libraries it needs separately for Tcl and for Tk. Problems like this would have been evident had Tk been delivered with other shared libraries. In fact, there are many parts of Tcl that could be delivered as separate libraries – socket and channel support are two examples. The reluctance of the Tcl developers to use their own library/package management system has allowed that section of Tcl to be weaker than it would have otherwise been.

A different kind of problem is illustrated by Tcl's idea of main. Originally, Tcl had no main. There was no need for it – Tcl was simply a library after all. But people wanted to write pure Tcl programs – perhaps with the tiniest of extensions. Since this was so prevalent, a main was added to the library. The drawback, of course, was that although Tcl's main satisfied many people, there was always something it didn't quite do – or do correctly. It changed frequently and became an annoyance to extension writers who wanted to rely on it. In contrast, Expect just used its own main, avoiding the problem entirely.

Problems Successfully Fallen Into

Earlier I mentioned that Tcl began providing a regular expression engine (mid '91). To Expect users, it seemed as if Expect simply leveraged that. In fact, Expect's pattern matching needs are not met by the regexp engine. For example, the interact command must be able to report not only 'match' and 'no match' but also whether a pattern *could* match if more characters arrive. Several other pattern matching capabilities are required by Expect and for this reason, Expect includes a complete reimplementation of Tcl's regexp engine as well as Tcl's glob pattern matcher.

The Tcl debugger shows a similar example of this phenomenon. Its ability to allow users to move up and down the stack is implemented by providing a complete reimplementation of TclGetFrame.

Fortunately these types of code rewrites didn't interfere with Tcl itself although they often required dependencies on TclInt.h as well as Tcl's source. The obvious risk here is that they could break at any time and with no recourse.

The I/O driver mechanism in Tcl demonstrates yet another slant on this problem. Expect requires more sophisticated buffering than Tcl offers. Thus, Expect had to provide its own file/buffer system. Expect doesn't use any of Tcl's I/O commands (e.g., read, gets, puts) and Tcl's buffering only serves to slow down Expect's I/O.

Documentation

I considered documentation crucial to Expect. The whole point of the software was so that people would not have to reinvent the automation wheel – and good documentation was a natural extension of that idea – people shouldn't have to figure out Expect either. And although the basics of Expect were intuitive (but then I *would* think so), many novel applications and aspects of Expect were quite non-intuitive and really needed examples and explanations.

The Expect book was a significant investment in my time – about three years from start to final publication [Libes95]. Official NIST funding was not available so I took it on as a personal activity. Much of the first year was spent waiting for permission from NIST and its parent organization, the US Department of Commerce. During that time, it wasn't even obvious that they would allow me to write such a book. But I desperately felt one was necessary. I was inundated by questions and requests for more information, despite having written almost a dozen technical papers for various journals and conferences. (Oddly, this is my first Expect paper at the Tcl/Tk workshop.) I also thought the book would bring a sense of closure to the work – a good theory even if it wasn't true – while at the same time making a contribution to a relatively new and significant field, interaction automation, that hadn't been formally recognized with any other books.

The book also had some other nice effects worth mentioning because they aren't at all evident from reading it. First, writing down an explanation often forces an author to reexamine implementation decisions. There were several aspects of Expect that I rewrote after I realized that my explanations in the book drafts were overly complex for no good reason. These rewrites delayed the book by many months but I was happy each time even though it meant rewriting both code and text in several chapters. The delays in the book were also fortuitous in that Tcl itself was changing. Although Tcl has continued to evolve, the bulk of the book is still accurate. In fact, the only piece of the book that should now be ignored is the section in the last chapter on Expect's timestamp command – this has been superseded by Tcl's clock command. (Expect continues to support timestamp for backward compatibility but the command is officially deprecated.)

The book had other consequences. For instance, it required me to learn more about Tcl. I felt obliged to be accurate about what I wrote and not gloss over things just because I didn't understand them. I also included a tutorial on Tcl itself. Not only were there no good ones at the time, but I still felt that Expect might be considered by many as a stand-alone application for people with no knowledge of (or interest in) Tcl. At the same time, I thought it was important to Tcl itself that it should have a book not written by the author of Tcl, to show that it was a language usable by and significant to others, and to show a set of practical applications.

Writing quality code and man pages is only half the battle. I encourage extension writers as well as application programmers to document their work through web pages and other online documentation, classes, papers, and books. We desperately need more.

Surprises

Many of my experiences with Tcl involved surprises. For instance, I was surprised that Tcl caught on – in the sense that people would write papers and have extended philosophical debates about it. When I began using it for Expect, it seemed irrelevant that no one had used Tcl – or might never use it for any other reason. Tcl looked like such a natural fit that I assumed people would not require any big effort to use Expect. At first, most of the scripts people wrote were just a dozen or so lines. It's hard to imagine how any language could further simplify statements such as:

```
expect "Login: "
send "$username\r"
```

In comparison, Expect rewrites in other languages such as Perl and C are much more clumsy. Although other languages have their advantages for various tasks, Expect is one of the few tools that hasn't been replicated as cleanly elsewhere. I've attempted or assisted such work with six different languages.

Due to constant request, I subsetted the Expect core into a Tcl-less library that could be integrated into other languages such as C and C++. I wasn't happy doing that – I've never written an Expect program that uses C for control. What's the point? Expect itself is neither memory- nor CPU-bound. Most of the time is spent waiting for the spawned program to respond. So Expect doesn't buy much in a compiled language. Interpretable languages like Python and Lisp make much more sense for high-level control tasks [Rossum][Mayer]. Indeed, the Expect library has been successfully integrated with both of these languages.

My philosophies diverged with Tcl in many other ways. For instance, as I had been trained, my early Expect code carefully checked for memory allocation failures. I kept that up well after finding out about Ousterhout's *memory is endless and if it's not, we're all in trouble* policy. Perhaps a year later, I finally ripped it all out. This left my code much more readable yet I felt quite uneasy for some time after. (The standalone Expect library continues to do memory allocation checking since the library can be embedded in other languages and systems that don't share the same policy.)

I was also surprised by users. People put Expect to use in ways I could never have imagined myself, making my own claims for Expect's application seem downright pedestrian. On the other hand, I was occasionally stunned by mail from Tcl users telling me that they couldn't use Expect because it was an extension and their management didn't allow them to use extensions with Tcl.

Change is Hell

Being on the leading edge is painful – change never stops. While Expect is no longer on the leading edge, it is definitely still subject to change. There is still a list of requests for 'improvements'. And a lot of these are understandable – such as porting it to the Windows and Macintosh platforms.

Other changes demand near instant response. Major changes to Tcl can mean major work for me. Sometimes even the most minor changes can mean major work for extension writers. For example, new instances of tclConfig.sh often require substantial study of and revision to Expect's configuration suite.

I have great sympathy for extensions that required changes to the core, such as iTcl [McLennan]. I successfully avoided that trap, although in the case of the debugger, only by sacrificing significant functionality – leaving users without access to line numbers.

Almost as bad however is Expect's use of Tcl's private undocumented interfaces. Such interfaces were often the only way to solve a problem. And since they rarely changed, their impact was insignificant. In fact, the major difficulty was simply listening to people complain about how TclInt.h wasn't available thereby preventing successful compilation of Expect. Distributing TclInt.h with Expect was too risky as it is full of magic numbers that change capriciously.

Lest my observations appear to belittle the Tcl developers, I will say it outright: It is clear to me that they take seriously the changes that impact extensions. Considering Tcl's recent capabilities and improvements, the Tcl developers have done a good job of minimizing the impact.

Clearly, change is a question that weighs heavily on all our minds. When is it acceptable to make incompatible changes? When Cygnus told me that they had Expect suites which performed over a million tests, I realized that there were likely others in similar position and that further changes could no longer be done so capriciously as had occurred in the past. This turned out to be just as well since publication of the book was yet another clear reason not to make any more changes.

Despite the relative stability of Expect's user interface for the past several years, it is likely that Expect will change, partly in order to incorporate ports to the Windows and Mac platforms. It will be very tempting to also try and correct the remaining deficiencies in Expect's user interface at the same time. Even ignoring corrections, Tcl's "reinventions" (especially I/O channels, binary I/O, and multithreading) can only be fully leveraged by redesigning Expect throughout – both the internals and user interface must change. The unnerving aspect is not the amount of work this will take, but the likelihood that future Tcl innovations may require yet more changes.

Concluding Notes

A project, not funded nor planned for the seven year effort it took, necessarily has many chapters to tell and some of them make sense only long after their occurrence. There are many more stories to Tcl – I'm limited here to present only a few. So I've selected stories that illustrate successes as well as stories that illustrate failures. We need both to learn from. And I implore educators who cite this paper not to pull lessons out of context.

Perhaps the most significant lesson of this paper is that what seems like a stable and dependable extension has required a tremendous amount of effort just to keep it working. Not only should the Tcl developers and the vendors appreciate the difficulties they cause but so should extension writers.

Expect originally started out as a small tool. It is still possible to use it that way. But Expect now features over 40 commands and requires a 600-page book to completely describe it and its applications. Like Tcl, Expect has grown. However, I often feel that control is out of my hands. Many of the changes in Expect have been forced by changes to Tcl. While in some cases, a change in Tcl has acted as an enabler, other times it has merely been a stumbling block. Expect's use of Tcl is especially worth critical examination because so many of the underlying assumptions originally motivating the choice of Tcl have changed. On the other hand, Expect's purposes have also changed, due in large part to Tcl's expanded capabilities. While the cost in tracking Tcl has been high, for the most part, Expect has benefited from it proportionally.

Acknowledgments

Thanks to George Howlett, Michael McLennan, David Beazley, Josh Lubell, Przemek Klosowski, Kari Harper, Howard Bloom, Rick Jackson, Chris Kuyatt, and Steve

Ray for reviewing this paper and making suggestions which dramatically improved it.

References

[Dalrymple] Dalrymple, Michael J., "User's Guide To NCFTP And The FTP Protocol", Colorado Center For Astrodynamics Research, University of Colorado, Boulder, CO, undated.

[Diekhans] Diekhans, Mark and Lehenbauer, Karl, "TclX – Extended Tcl", http://www.neosoft.com/tcl/TclX.html.

[Friedl] Friedl, Jeffrey E.F., "Mastering Regular Expressions", O'Reilly and Associates, January 1997.

[Howlett] Howlett, George A., "BLT", http://www.tcltk.com/blt/index.html.

[Libes91] Libes, Don, "Expect: Scripts for Controlling Interactive Processes," Computing Systems, Vol. 4, No. 2, University of California Press Journals, Spring 1991.

[Libes93] Libes, Don, "A Debugger for Tcl Applications," Proceedings of the 1993 Tcl/Tk Workshop, Berkeley, CA, June 10-11, 1993.

[Libes95] Libes, Don, "Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs", O'Reilly and Associates, January 1995.

[Libes97] Libes, Don, "Managing Tcl's Namespaces Collaboratively", Proceedings of the 1997 Tcl/Tk Workshop, Boston, MA, July 14-17, 1997.

[Mayer] Mayer, Neils P., "The OSF/Motif Widget Interpreter", http://www.eit.com/software/winterp/doc/winterp.doc, July 24, 1994.

[McLennan] McLennan, Michael, "[incr tcl] – Object-Oriented Programming in Tcl, Proceedings of the 1993 Tcl/Tk Workshop, Berkeley, CA, June 10-11, 1993.

[Nowitz] Nowitz, D. A., "UUCP Implementation Description", UNIX Programmer's Manual, Section 2, AT&T Bell Laboratories.

[Ousterhout] Ousterhout, John. K., "Tcl and the Tk Toolkit", Addison-Wesley, 1994.

[Phelps] Phelps, Thomas A., "Two Years with TkMan: Lessons and Innovations, Tcl/Tk Workshop, Toronto, Canada, July 6-8, 1995.

[Poindexter] Poindexter, Tom, "Oratcl", Tcl/Tk Extensions, ed., Mark Harrison, O'Reilly & Associates, to appear.

[POSIX] International Organization for Standardization, International Electrotechnical Commission, "ISO/IEC 9945, Portable Operating System Interface (POSIX)", IEEE, New York, NY, 1990.

[Rossum] Rossum, Guido v., Python Language Home Page, http://www.python.org.

[Stein] Stein, L., "CGI.pm: A Perl Module for Creating Dynamic HTML Documents with CGI Scripts", SANS 96, May '96.