expect: Curing Those Uncontrollable Fits of Interaction

Don Libes

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

ABSTRACT

UNIX programs used to be designed so that they could be connected with pipes created by a shell. This paradigm is insufficient when dealing with many modern programs that *demand* to be used interactively.

expect is a program designed to control interactive programs. **expect** reads a script that resembles the dialogue itself but which may include multiple paths through it. Scripts include:

- 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 successfully deals with interactive programs. It also solves several other large classes of problems which UNIX shells do not.

Keywords: expect, interaction, programmed dialogue, shell, Tcl, UNIX, uucp

Reprinted from *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, California, June, 1990.

January 21, 1992

1 Introduction

UNIX programs used to be designed so that they could be connected with pipes created by a shell. This paradigm is insufficient when dealing with many modern programs that *demand* to be used interactively.

For example, the **passwd** program is used to change passwords. **passwd** prompts for the password. There is no provision for passing the information any other way. This means that you cannot write a shell script which uses **passwd** without letting it do the prompting and reading. Thus, it is impossible to write a script that, say, rejects passwords that are in the system dictionary.[1]

This illustrates one type of difficulty in the user interface provided by shells such as **sh**, **csh**, **ksh** and others (which I will generically refer to as "the shell" in the rest of the paper). I will discuss several other difficulties later. All of them have to do with the shell's inability to communicate with interactive programs. My solution is called "**expect**".

expect is a program that "talks" to other interactive programs according to a script. By following the script, **expect** knows what can be expected from a program and what the correct responses should be. An interpreted language provides branching and high-level control structures to direct the dialogue. In addition, the user can take control and interact directly when desired, afterward returning control to the script.

The name "expect" comes from the idea of *send/expect* sequences [4] popularized by **uucp**, **kermit** and other communications programs. However, unlike these programs, **expect** is generalized so that it can be run as a user-level command with any program and task in mind. (**expect** can actually talk to several programs at the same time.)

Using **expect**, it is possible to create a script that solves the **passwd** problem. Here are some other things **expect** can do, each requiring only a small amount of script:

- Have your computer dial you back, so that you can login without paying for the call.
- Start a game (e.g., **rogue**) and if the optimal configuration does not appear, restart it (again and again) until it does, then hand over control to you.
- Run **fsck**, and in response to its questions, answer "`yes`", "`no`" or give control back to you.
- Connect to another network or BBS (e.g., MCI Mail, CompuServe) and automatically retrieve your mail so that it appears as if it was originally sent to your local system.

Although these problems are conceptually simple, none of them can be solved by the shell. What is wrong? What do we do with the hard cases!?!

1. Ironically, the original reason for having **passwd** perform the prompting was for security!

2 Solving the problem

Since I am claiming that **expect** can solve problems the shell cannot, let us first start by reviewing what the shell is capable of.

Each process created by the shell is given a standard input (stdin), standard output (stdout) and standard error (stderr) (although I will ignore the last one for now). The shell can connect these to other processes or files. However, shell pipes and redirection are purely one-way. Ritchie [7] has described shell pipe notation as "*unabashedly linear*". There is no shell notation to create two processes which have their standard input and output cross-connected (Figure 1).

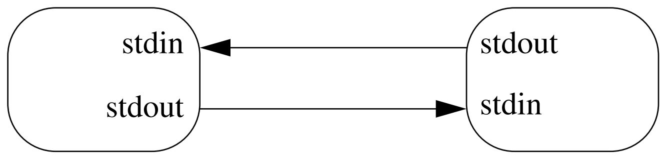

Figure 1. The shell cannot connect two processes in this way.

The traditional notation *proc1* | *proc2* indicates that output flows from *proc1* to *proc2*. There is no data flow from *proc2* to *proc1*. Indeed, the only entity that the shell can interact with on a 2-way basis is the user. Viewed from the opposite direction, only the user is capable of 2-way interaction with programs.

Since the shell cannot construct such connections nor can it participate in them, it cannot "talk" to interactive programs. This prevents any program from interacting with another unless both have been specially designed to do so. This is the first problem.

A second problem is that the shell has no way to prevent a program from bypassing the standard input and output conventions. Programs are free to open **/dev/tty** to communicate directly with the user. This is often used to bypass shell redirection. For example, **crypt** does this because its input is redirected while it interactively demands an encryption password.[2]

The first problem is easy to solve. A program is created (by the shell) which internally spawns the process to be controlled. The spawned process is established so that its standard input and standard output remain connected to the original program (Figure 2).

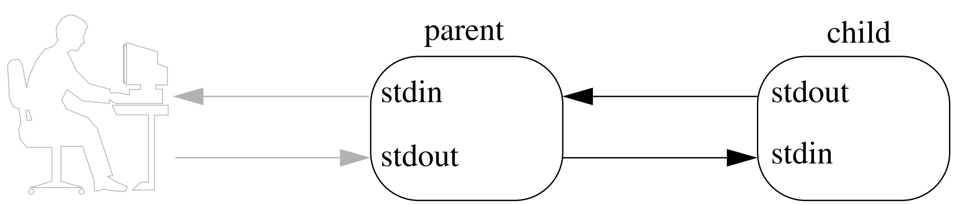

Figure 2. The parent creates a child process so that it can read the child's standard input and write its standard output. The user is left connected by the shell, but plays no role here.

2. Again, this was done for security reasons.

The parent now reads a command script. When it finds **send** commands, it sends data to the child. When it finds **expect** commands, it watches the output of the child for a pattern. If the pattern is matched, the parent goes to the next command in the script (Figure 3).

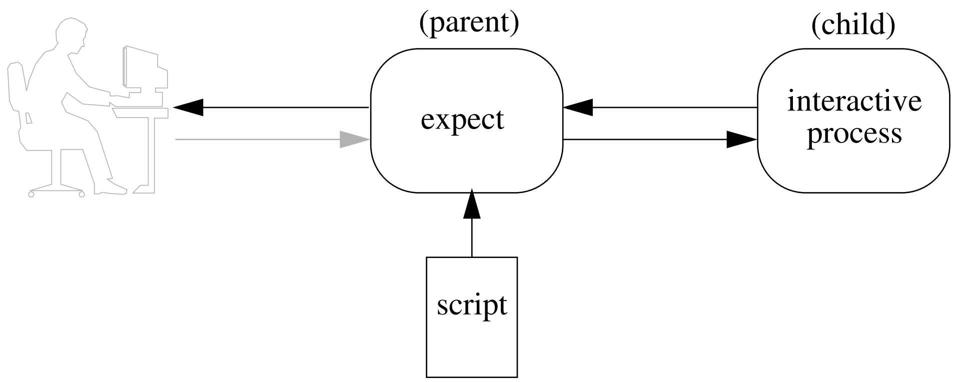

Figure 3. **expect** "talks" to the child process, according to the script. Interaction is copied to the user terminal and appears as if the user actually typed it.

When an **interact** command is found, the parent simply copies characters from the user to the child and vice versa (Figure 4). Control may be returned to the script at the user's convenience.

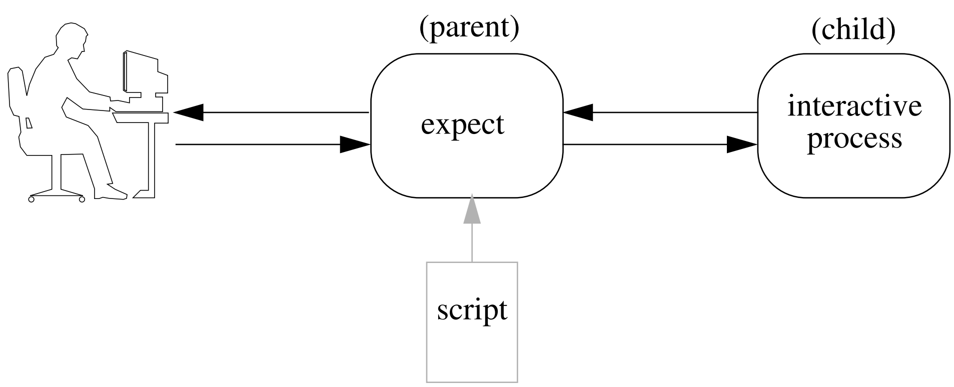

Figure 4. In **interact** mode, the user takes control and types directly to the child process.

These simple ideas are the heart of **expect**. However, a few refinements allow **expect** to address several more problems.

2.1 Pseudo-terminals

Pipes do not support terminal semantics. For example, programs such as **rogue**, **emacs**, etc., which require the terminal size in order to run, will not run over pipes. Thus, **expect** uses pseudo-terminals (ptys). Ptys are logical device drivers that give the "look and feel" of real terminal drivers despite the fact that they are used to connect two processes together. In addition, ptys solve the **/dev/tty** problem noted above. Programs that open **/dev/tty** will actually end up speaking to their pty.

Ptys support two paths of communication flow (much like two pairs of pipes). What happens to the standard error? It is overloaded into the path already used by standard output. The rationale is that if a user sitting at a terminal can make decisions based on a common output stream of both standard output and the standard error, then so can the script.

Just as the user may redirect either standard input or standard output, so may the script. But by default, neither is redirected. This is true for both **expect** and the shell.

2.2 Job control

The script may interact with a number of processes simultaneously. Figure 5 shows several processes being controlled at the same time. To control multiple processes, a user would use job control. So does **expect**. Script job control is actually easier to use because it can be programmed whereas shell job control must be hand entered.

For example, suppose you are using **csh** and have to type something at *proc2* depending upon what *proc1* is telling you. If this happens 100 times, you have to type **^Z/fg** sequences 200 times[3]! With **expect**, a simple loop suffices. A good example is to try and connect two Eliza [10] or **chess** processes to each other. Remember that the output of the UNIX **chess** program is not directly usable as input.

The user can also be manipulated as if they were a process. The effect is exactly like a shell script reading from and writing to the user. In other words, the source of I/O to the user is the script rather than an underlying interactive process. This is illustrated by the user appearing alongside the processes.

If desired, the user can take control (appearing on the left side of the figure) and enter commands just like the script. Both the script and the user can take control from and return it to each other. In the figure, the two have been moved closer together to emphasize the near equal relationship of them.

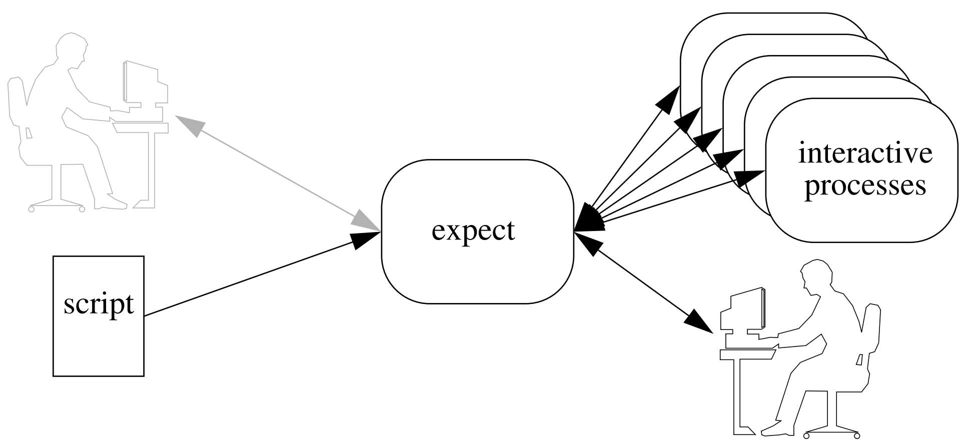

Figure 5. **expect** is communicating with 5 processes simultaneously. The script is in control and has disabled logging to the user. The user only sees what the script says to send and is essentially treated as just another process.

3. Even if you were using a window system, you would still have to cut and paste 100 times.

2.3 High-level language

The last important feature of **expect** is the language itself. It is described in the next section.

3 expect scripts – What do they look like?

expect scripts are written using a high-level procedural language. The language is interpreted and resembles the shell in many ways. Elements are also derived from C and LISP. Despite its mixed heritage, much of the excess baggage from these other languages has been omitted leaving a modest but capable language. The language consists of a core of features called *Tcl* (Tool Command Language) and is described by Ousterhout [5]. This section will only give a brief overview and enough details to describe the sample **expect** scripts later on.

The Tcl core consists of control flow statements such as **if**, **while**, and **case**. Tcl supports procedure definition, recursion, scoping, and more. UNIX programs may be called and files manipulated. Expression evaluation is provided by a small set of primitives that manipulate the only type – strings. (Conversion to and from other types is performed automatically, a la SNOBOL.)

The following Tcl fragment (from [5]), swaps the values of variables **a** and **b**, if **a** is less than **b**.

```
if {$a < $b} {
        set tmp $a
        set a $b
        set b $tmp
}
```

Here is a command to define a recursive factorial procedure:

```
proc fac x {
        if {$x == 1} {return 1}
        return [expr {$x * [fac [expr $x-1]]}]
}
```

The syntax and semantics are sufficiently close to C and the shell that the meaning of these examples should be intuitively obvious. For lack of space, I will not describe Tcl further, however it is completely described by Ousterhout [6]. For that matter, it is not particularly germane to the theory of **expect**. Indeed, Ousterhout [5] makes the point that the "*syntax of the Tcl language is unimportant: any programming language*" could provide similar features. The salient features of Tcl are that it is:

- simple – It is expected that most Tcl programs will be short.
- programmable – Tcl applications are general-purpose and are not known in advance.
- efficiently interpreted – Tcl must be able to execute commands quickly enough that user interaction is not noticably impeded.
- internally interfaceable to C – Tcl must allow one to add new commands that work synergistically with existing Tcl commands.

As the last bullet says, Tcl is designed to allow the addition of new commands. **expect** adds twelve commands to the Tcl language. I will now present the more interesting of these new commands.

3.1 Interaction commands

send *args*

sends *args* to the current process. Strings are interpreted following Tcl rules. For example, the command

```
send hello world\r
```

sends the characters, `h e l l o` <space> `w o r l d` <return> to the current process.

expect *patlist1 action1 patlist2 action2 . . .*

waits until the output of the current process matches a pattern, or a specified time period has passed. Each *patlist* consists of a single pattern or list of patterns. If a pattern is matched, the corresponding action is executed. The result of the action is returned from **expect**. The exact string matched (or read but unmatched, if a timeout occurred) is stored in the variable **expect_-match**. If ***patlist*** is **eof** or **timeout**, the corresponding action is executed upon end-of-file or timeout, respectively. The default timeout period is 10 seconds but may, for example, be set to 30 by the command `set timeout 30`.

The following fragment is from a script that involves a login. **abort** is a procedure defined elsewhere in the script, while the other actions use Tcl primitives similar to their C namesakes.

```
expect {*welcome*}  break \
       {*busy*}     {print busy; continue} \
       {*failed*}   abort \
       timeout      abort
```

Patterns are the usual C-shell-style regular expressions. Patterns must match the entire output of the current process since the previous **expect** or **interact** (hence the reason most are surrounded by the ***** wildcard). However, more than 2000 bytes of output can force earlier bytes to be "forgotten". This may be changed by setting the variable **match_max**.

interact [*escape-character*]

gives control to the user. User keystrokes are sent to the current process, and the standard output and standard error of the current process are returned to the user. Any valid script commands may be entered after pressing the optional *escape-character*. Control is returned to **interact** if the **continue** command is entered. If the **return** command is entered, **interact** immediately returns with the argument of **return** (or the empty string if none is given).

During **interact** (except when entering commands via the escape character), job control is disabled so that all characters may be passed to the current process.

3.2 Job control commands

close

closes the connection to the current process. Most interactive programs will detect EOF on their standard input and exit; thus **close** usually suffices to kill the process as well. Both **expect** and **interact** will detect when the current process exits and implicitly do a **close**.

spawn *program* [*args*]

creates a new process running *program args*. Its standard input, standard output, and standard error are connected to **expect**, so that they may be read and written by other **expect** commands. The connection is broken by **close** or if the process itself closes any of the file descriptors.

When a process is started by **spawn**, the variable **spawn_id** is set to a descriptor referring to that process. The process described by **spawn_id** is considered *the current process*.

spawn returns the UNIX process id. Note that this is not equivalent to the descriptor in **spawn_id**.

Internally, **spawn** uses a pty, initialized the same way as the user's tty. When this is not possible (such as when **expect** has no controlling terminal), **spawn** uses the default pty settings. If these are not appropriate, the user can spawn a shell, set the pty parameters directly, and then **send** (rather than **spawn**) the original command to the shell.

select *spawn_id1 spawn_id2 ...*

returns a subset of the given *spawn_id*s that have input pending. **select** waits until at least one spawn_id can be read or until the timeout (see **expect** command above) has expired.

There is no explicit command to switch jobs. Rather, the variable **spawn_id** determines the current process. **spawn** sets this as a side-effect so that a script interacting with only one process need not ever mention **spawn_id**. **spawn_id** may be read and written through Tcl's **set** command.

Here is an example showing how job control could be used to have two **chess** processes interact. After spawning them, one move is sent by hand to get things started. In a loop, a move is sent from one process to the other, and vice versa. The **read_move** and **send_move** procedures are left as an exercise for the reader. (They are actually very easy to write, but too long to include here.)

```
spawn chess
set chess1 $spawn_id
spawn chess
set chess2 $spawn_id
# force someone to go first
send p/k2-k3
for {} {1} {} {
        read_move
        set spawn_id $chess1
        send_move
        read_move
        set spawn_id $chess2
        send_move
```

```
}
```

There is no command (analogous to **csh**'s **bg**) to let processes run without interaction, since presumably input is a necessary part of interaction and cannot be supplied in the background. Processes that enter into a lengthy phase during which no input takes place will free run by default, although output will eventually clog the pty if not periodically flushed. Fortunately, this is easy to do.

3.3 Miscellaneous commands

The remaining commands will not be described in detail. For complete descriptions, see Libes [2]. These miscellaneous commands fall into the following classes:

- tracing - Programs may be traced to assist debugging.
- exiting - **expect** can return an exit code, allowing it to be intelligently used in shell scripts.
- logging - Logging to files and/or the user terminal is flexible. This allows interaction with the user while hiding some or all of the interaction taking place with programs.

4 More examples

expect scripts are similar in style to shell scripts. They look like the interaction they are supposed to control. Just as shell scripts primarily consist of the commands as a user might type them, **expect** scripts consist primarily of the interaction that a user might see.

Here are two examples. The first runs the BSD adventure game **rogue** repeatedly until a configuration with unusually good attributes (i.e., strength of 18) appears, after which control is given to the user.

```
# rogue.exp - find a good game of rogue
set timeout 3
for {} 1 {} {
        spawn rogue
        expect {*Str:\ 18*} break \
                timeout      close
}
interact
```

Some comments are in order: The first line is a comment, naming the file and explaining what it does. The second line sets a short timeout. This is appropriate since we are dealing with a local program that will respond very quickly.

for introduces a C-like **for** loop, with the same control arguments as in C. Here, the loop repeats forever. After **rogue** is started, we look for the text of interest in the output. **rogue** is a graphics program which uses curses. Curses does not guarantee screens are created in an intuitive manner, and **expect** programmers must understand that. However, it is not a problem here.

If **expect** does not find what it is looking for, the dialogue is terminated via **close** (which will cause **rogue** to go away) and the loop restarted. If the desired string is found, the loop is terminated and the user given control of the dialogue through **interact** on the last line.

The second example dials a phone. It can be used to reverse the charges, so that long-distance phone calls are charged to the computer. It is invoked as `expect callback.exp 12016442332` where the script is named **callback.exp** and +1 (201) 644-2332 is the phone number to be dialed. (Scripts may also be turned into executables on systems which support the **#!** magic.)

```
# first give the user some time to logout
exec sleep 4
spawn tip modem
expect {*connected*} {}
send ATZ\r
expect {*OK*} {}
send ATDT[index $argv 1]\r
# modem takes a while to connect
set timeout 60
expect {*CONNECT*} {}
```

The second line illustrates how a UNIX command with no interaction can be called. `sleep 4` will cause the program to block for four seconds, giving the user a chance to logout, since the modem will presumably call back to the same phone number that the user is already using.

After spawning **tip**, the modem dials the number. (The modem is assumed to be using Hayes protocol, but it would be easy to expand the script to handle others.) No matter what happens, **expect** terminates. If the call fails, it is possible for **expect** to retry, but that is not the point here. If the call succeeds, getty will detect DTR on the line after **expect** exits, and prompt the user with `login:`. (Actual scripts usually do more error checking.)

This script illustrates the use of command-line parameters, made available to the user as a list named **argv**. Commands may also be passed in and executed by the program. A special flag (**-c**) allows execution of commands before any in the script. For example, an **expect** script can be traced without reediting by invoking it as `expect -c "trace ..." script.exp` (where the ellipsis indicates a tracing option).

5 What classes of problems does expect address?

expect addresses a surprisingly large class of problems that the shell does not. At the same time, **expect** does not attempt to subsume functions already handled by other utilities. For example, there is no built-in file transfer capability, because **expect** can just call a program to do that.

The following categories are not meant to be disjoint but to sharpen the focus of examples that may share multiple problems.

5.1 Programs that demand interactivity

Programs that demand interactivity such as **passwd** and **tip** are easy to control with **expect**, but impossible with the shell. In fact, **tip** has special code to do dialing before a conversation, but it is quite limited in power. **expect** eliminates the need for this type of special code in many programs.

5.2 Programs that cross machine or program boundaries

Making a shell script run across machine boundaries is not possible except in limited ways. For example, shell scripts that involve telneting to another host cannot log in nor can they continue the shell script on the remote host. **expect** does not see these kinds of machine or program boundaries.

5.3 Programs that read and write /dev/tty

Programs that read and write **/dev/tty** cannot be used from shell scripts without the shell script accessing **/dev/tty**. **passwd**, **crypt**, and **su** are examples of programs that cannot be controlled by the shell but can by **expect**.

5.4 Programs that flush input

Some interactive programs believe they are doing the user a favor by flushing input after detecting an error. Particularly clever programs such as **rn** [9], not only flush input already received but continue to flush input for a short time afterwards to allow for communications or user delays.

Redirecting standard input from the shell is ineffective with such programs since there is no control over how much can be lost when input flushing occurs. **expect**, on the other hand, will wait for the desired prompt rather than proceeding to send commands blindly.

While it was not my intention, **expect** provides a foundation for a relentless password cracking tool. However, to show my compassion I will remind system adminstrators that one preventative measure is to lock out an account after a small number of incorrect passwords have been tried.

5.5 Passing control from user to/from script

Programs such as **rogue**, **tip**, **telnet**, and others have a frequently repeated, well-defined set of commands and another set that are not well-defined. For example, **telnet** is always started by logging in, after which the user can do anything. **expect** can pass control from the script to the user to provide this ability. In fact, **expect** can take control at any time to execute sequences of commonly repeated commands.

5.6 Ostensibly non-interactive programs

Many programs are ostensibly non-interactive. This means that they can be run from a shell script, but with greatly diminished functionality. For example, **fsck** can be run from a shell script only with the **-y** or **-n** options. The manual [1] defines the **-y** option as follows:

> *"Assume a yes response to all questions asked by fsck; this should be used with extreme caution, as it is a free license to continue, even after severe problems are encountered."*

The **-n** option has a similarly worthless meaning. This kind of interface is inexcusably bad, and yet many programs have the same style. For example, **ftp** has an option that disables interactive prompting so that it can be run from a script. But it provides no way to take alternative action should an error occur.

expect is useful with such programs. For example, it could be programmed to answer "`yes`" and "`no`" depending on the question from **fsck**. Control could be turned over to the user for questionable cases.

5.7 Programs with poorly written interfaces

Ousterhout [5] makes the observation that "a general purpose, programmable command language amplifies the power of a tool by allowing users to write programs in the command language in order to extend the tool's built-in facilities."

Few tools actually include such a language. Examples are shells and **emacs**. There are also a few that have more simplistic facilities such as the **.rc** files of **mail**, **vi**, and **dbx**. But in all, there are very few tools with the flexibility of a good language. Indeed, Tcl was designed to address this very problem.

Expecting UNIX tools to be rewritten using Tcl is a noble but unlikely proposition. However, a similar effect can be achieved with **expect**. For example, **expect** can be used to initialize a tool, much like a set of commands from an **rc** file. And like Tcl, **expect** presents a uniform language for doing so. In effect, **expect** provides a way of giving the power of Tcl to tools without any effort spent rewriting them.

If desired, **expect** can be run in the background, completely disassociated from user input. **expect** is capable of returning a status value to a script, often more meaningfully than the original tool or task. (Realistically, there is little reason for a program that originally could only have been run interactively to return a status of any type.)

5.8 Multiple programs never designed to work together

expect is capable of connecting programs that were not originally designed to be connected. In contrast to non-interactive filters that form pipelines, interactive programs have foresaken any attempt to be driven by another program. Eliza and **chess**, both mentioned earlier, are good examples.

A more complex example is communication with another network or bulletin board system. Commercial systems such as MCI Mail and CompuServe do not forward mail, expecting that users will dial up and read mail interactively. An **expect** script can dial up such a system and check for mail. If mail is found, a **mail** process can be started on the local system and fed input from the remote system. Mail will then appear as if it was originally mailed to the local system. Since **expect** can run in the background, this can be done at night, every hour, or whatever is convenient.

5.9 Dynamic and complex pipes and redirection

A number of projects have been built to step beyond the linearity of pipes enforced by the shell. Two notable examples are **gsh** and MTX.

gsh [3] is based on the Bourne shell, but handles graphs of processes, such as sending the output of one process to two processes, or building a set of three process in a cycle. While **expect** was not designed for this purpose, it can do this as a byproduct of its complete control of any dialogue. Of course, the result will not be as fast because **expect** necessarily interposes itself in order to control the dialogue.

MTX [8] is a screen-based pipe manager. It solves the same set of problems as **gsh**, although the interface is mouse-oriented instead of keyboard-oriented. In addition, MTX can rearrange connections in use. It does this using the same pty mechanism that **expect** does (with a similar penalty in throughput), although MTX does not provide any automated control of the dialogue.

In summary, **expect** can emulate dynamic and complex pipes and redirection. It is a simple matter to emulate processes of pipes in a graph. Automatic rearrangement is possible either under the control of a user or when signalled by data. Complex redirection such as arbitrary fan-out is also trivial and easily supercedes the capabilities of **tee**.

6 Is expect a shell?

The beginning of this paper compared the shell to **expect**. To repeat, the shell is incapable of the interactive dialogue that **expect** can perform. But how does **expect** compare with the shell? Is **expect** as powerful as a shell?

The base language of **expect**, Tcl, is quite powerful and is certainly capable of doing the same kinds of programming as the shell. Indeed, as a programming language, Tcl is functionally almost a replacement for any of **sh**, **csh** and **ksh**. One noticeably missing feature is redirection, which Tcl can perform by calling the shell.

For interactive use, however, Tcl has little support. There is no history or job control. This is understandable, considering that Tcl was not designed to be used interactively. The additional commands added by **expect** do not change that. Job control as supported by **expect** is strictly command-based while the shell offers interrupt keys as shorthand. Indeed, typing job control characters at **expect** itself only affects **expect**, not the other processes **expect** is interacting with. (The exception to this is that when in **interact**, job control characters are sent to the current process.)

While **expect** can be used interactively (by pressing the *escape-character* while in **interact**), it was not designed to be and lacks pleasant features such as history and interrupt key-based job control. Its interactive mode is seen as primarily useful for experimenting, although some very powerful results are possible by typing **expect** commands in directly, or indirectly by say, programmable function keys or software stream modules that perform interactive line editing.

expect can call the shell and be called by it. The two work very well together. An obvious question is if one can be subsumed into the other. Adding the **expect** commands to the shell is probably the easiest implementation, and I see little technical difficulty in doing so.

7 Implementation discussion

This section discusses areas of the implementation that are unusual in some way and might be educational to a large percentage of readers. The source code is quite readable and perusal of it is encouraged.

7.1 Command language

From the list of examples in section 5, it should be apparent that the functions provided by **expect** have long been desired. I began thinking about them several years ago and have experimented with various implementations. The biggest stumbling block was the language. I knew I needed one, I knew how to write one, but I wasn't sure how far to go nor was I particularly interested in the task of writing yet-another-utility-language.

Tcl was the solution: it was designed specifically as an embeddable language. Tcl comes with a core of commands to which the application writer can add application-dependent commands. Adding the **expect** commands was relatively painless although a number of different command designs were tried before being finalized.

In my environment (Sun 3 running Sun OS 4.0.3), the Tcl library (version 2.1) is approximately 8000 lines, including comments (45k object code); the additional **expect** source (version 1.7) is 1700 lines (13k object code). Clearly, the Tcl code dominates **expect**. Here, **expect** is a wrapper around Tcl, which is probably different than how the original Tcl designers foresaw its role.

Tcl is not the only possible base for **expect**-like functionality. Prior to Tcl, I looked at the send/expect control used by **uucp**, **kermit**, and other communication programs. However, these are quite primitive and do not even provide adequate flexibility for their own tasks. For example, system administrators always embed calls to **uucp** in shell scripts which can repeat dialing upon failure.

Two alternatives to Tcl that I could have chosen are **emacs** and **perl**. While not specifically designed to provide a language for tools, both systems have embedded interpreters that can be used this way. Unfortunately, the tradeoffs are numerous and deserve more space than I can devote here. It would probably be worth doing an implementation in each to compare them.

7.2 Multisource asynchronous I/O

expect requires multisource asynchronous I/O primarily for the **interact** command (which listens for characters from the user and a process at the same time). Using this feature requires different implementations on different UNIX systems is probably the least portable part of the software.

Berkeley UNIX has long supported the **select** system call which permits waiting for activity from a set of file descriptors. Thus, virtually all BSD-derived UNIX systems support **select**. On the other hand, System V has only recently supported an equivalent of **select**. Release 3 supports **poll** which is comparable to **select**; however, a large percentage of SV systems are still based on Release 2, which forces applications to poll by busy-waiting. (POSIX has not yet provided a means of performing multisource asynchronous I/O, though it seems inevitable.) Earlier systems, such as V7, could not even poll. Typically, programs such as **uucp** forked an auxiliary process which blocked waiting for data from one direction while the original process blocked waiting for data from the other direction. I consider this untenable, since some **expect** applications spawn many processes, both sequentially and in parallel. Using this style of communication requires two extra utility processes for each real process requested by the user. For example, Figure 5 would need 12 more processes than it does in the current implementation.

For lack of more sophisticated code (and being not at all clear that it is even possible), the implementation cannot simultaneously interact with multiple processes on systems lacking **select**, **poll**, or something similar.

7.3 Job control

This section discusses the interaction between BSD-style job control and **expect**. **expect** has its own way of controlling jobs, discussed extensively in section 2.2.

In a sense, **expect** finesses the problem of job control. When sitting at the keyboard running **expect**, a user may perform job control (i.e., by pressing job control characters), which will affect the **expect** program just like any UNIX program. For example, by default, a `^Z` will stop **expect**.

An exception to this is during the **interact** command. Since **expect** has no idea if its client programs are interested in seeing job control characters, all characters (except an optional escape character) are passed through to the current process. Thus, programs that run in raw mode (e.g. **rogue**) and programs that handle job control themselves (e.g., **csh**) can run with their full functionality.

Of course, users choosing to interact directly must understand that programs which do not handle job control signals can be unpleasant. For example, sending a `^Z` to a program that does not catch **SIGSTOP** will cause it to be stopped by the kernel. But since there is no program behind it to catch control, **expect** will wait as well. The program can be resumed by sending a **SIGCONT**, but this is presumably inconvenient. The moral is, when dealing with programs that do not understand job control, either do not send them job control signals or place a shell behind them that does.

Switching jobs internal to **expect** from within **interact** can be done without needing to back every process with a shell. The technique is to escape to the **expect** interpreter (by pressing **interact**'s escape character), set **spawn_id** to the desired process, and return to **interact** (via **return**). The user will now be interacting with the desired process.

7.4 Throughput

While throughput statistics are *de rigueur* in USENIX implementation papers, it is difficult to quantify performance in this type of program. For example, how does one compare **expect**'s overall impact on system throughput to that introduced by canonical input processing of typical human typing? About the only thing that is clear is that **expect** uses a fraction of the real time that a user does.

Internally, **expect** processing is heavily dependent upon scripts. For example, the **rogue** script presented earlier examines about 10 games per second (and is fun to watch). Most of the real time is spent waiting for the game itself. Of the CPU time, about 40% is spent pattern matching to guide the script, 26% in I/O, 16% in **open**, **close**, and **ioctl**, 8% in **fork**, and 5% in timer calls. The large amount of time spent in **open**, **close**, and **ioctl** are due to the inefficient technique required by BSD UNIX to locate and initialize ptys.

The **rogue** script is so biased towards short dialogues, it is likely that most other scripts will spend even larger percentages of time pattern matching. While guiding the dialogue is a primary function, this indicates an area for improvement. The current (Tcl-supplied) pattern matcher could be improved, for example, by compiling patterns. But **expect** has even more demands. Regular expression pattern matching is performed on input each time a **read** completes. If characters arrive slowly, the pattern matcher scans the same data many times. System indigestion can play a big role here, as larger scheduling quanta drive the pattern matcher less frequently when more characters at a time are gathered from each pty. The performance of a pattern matcher that does not need to rescan over earlier data needs to be studied.

8 Comments and conclusions

The UNIX shell paradigms are incapable of intelligently managing interactive programs. This has been a long-standing problem, traditionally solved by avoidance. Yet the number of interactive programs grows daily, and shells have not changed to address this.

expect solves these problems directly and with elegance. **expect** scripts are small and simple for problems that are small and simple. While I am not so naive to believe all **expect** scripts will be small, it is apparent that the scripts scale well. They are comparable in style to shell scripts, being task-oriented, and provide synergy with shell scripts, both because they can call shell scripts and be called by them. It would be a worthwhile experiment to marry the features of **expect** to the shell, and I see little technical difficulty in doing so.

Some interesting open questions remain: How would the buffering work in a combined **expect/select** command. If **expect** had a built-in terminal emulator, could one look for "regions" of character graphics? Lastly, how could **expect** emulate interactions with window systems, such as mousing and dragging. Each of these requires further research. Nonetheless, as long as there are shells, there will be interactive programs that are not controllable by them, and **expect** will continue to be useful.

9 Acknowledgements

This work was jointly funded by the NIST Automated Manufacturing Research Facility (AMRF, project 734-3385) and Scientific and Technical Research Services (STRS, project 734-3106).

Thanks to Scott Paisley for writing a program called **stelnet** (**s**mart **telnet**). **stelnet** ran **telnet** and performed a simple send/expect conversation to login. **stelnet** had only straight-line control without error processing, used pipes instead of ptys, and lacked pattern matching and job control.

Nonetheless, **stelnet** proved very useful in the AMRF, and got me to thinking about a more generic tool.

John Ousterhout is responsible for Tcl, without which **expect** would not have been written. John also critiqued **expect** as well as this paper. I am indebted to him.

Several people made important observations or wrote early scripts while I was still developing the command semantics. Thanks to Rob Densock, Ken Manheimer, Eric Newton, Scott Paisley, Steve Ray, Sandy Ressler, and Barry Warsaw. And also, like, thanks to my grammarians, Scott Bodarky, Ted Hopp, and Sue Mulroney, who read read and corrected everry sentence in the paper but like this one, right.

10 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**.

11 References

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

[2] Don Libes, "*expect(1) – programmatic dialogue with interactive programs*", unpublished manual page, National Institute of Standards and Technology, February, 1989.

[3] Chris McDonald and Trevor Dix, "Support for Graphs of Processes in a Command Interpreter", *Software: Practice & Experience*, Volume 18 Number 10, p. 1011-1016, October 1988.

[4] D. Nowitz, "*Uucp Implementation Description*", UNIX Programmer's Manual, Bell Laboratories, October, 1978.

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

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

[7] Dennis Ritchie and Ken Thompson, "The UNIX time-sharing system", *Communications of the ACM*, Volume 17, Number 7, 635-375 (1974).

[8] Stephen Uhler, "*MTX – A Shell that Permits Dynamic Rearrangement of Process Connections and Windows*", Proceedings of the Winter 1990 USENIX Conference, Washington, D.C., January 22-26, 1990.

[9] Larry Wall, "*rn(1) – read news program*", unpublished manual page, May 1985.

[10] Joseph Weizenbaum, “Eliza – A Computer Program for the Study of Natural Language Communication between Man and Machine”, *Communications of the ACM*, Volume 9, Number 1, January 1966, p. 36-45.

photo

Don Libes
National Institute of Standards and Technology

Don Libes received a B.A. in Mathematics from Rutgers University and an M.S. in Computer Science from the University of Rochester.

Currently at the National Institute of Standards and Technology, Don is engaged in research that will help U.S. industry measure the standard hack. Unfortunately, NIST does not have a very good sense of humor, so he was forced to write his first book