Tcl: An Embeddable Command Language

John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720
ouster@sprite.berkeley.edu

ABSTRACT

Tcl is an interpreter for a tool command language. It consists of a library package that is embedded in tools (such as editors, debuggers, etc.) as the basic command interpreter. Tcl provides (a) a parser for a simple textual command language, (b) a collection of built-in utility commands, and (c) a C interface that tools use to augment the built-in commands with tool-specific commands. Tcl is particularly attractive when integrated with the widget library of a window system: it increases the programmability of the widgets by providing mechanisms for variables, procedures, expressions, etc; it allows users to program both the appearance and the actions of widgets; and it offers a simple but powerful communication mechanism between interactive programs.

This paper will appear in the 1990 Winter USENIX Conference Proceedings

---

The work described here was supported in part by the National Science Foundation under Grant ECS-8351961.

1. Introduction

Tcl stands for ''tool command language''. It consists of a library package that programs can use as the basis for their command languages. The development of Tcl was motivated by two observations. The first observation is 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. Among the best-known examples of powerful command languages are those of the UNIX shells [5] and the Emacs editor [8]. In each case a computing environment of unusual power has arisen, in large part because of the availability of a programmable command language.

The second motivating observation is that the number of interactive applications is increasing. In the timesharing environments of the late 1970's and early 1980's almost all programs were batch-oriented. They were typically invoked using an interactive command shell. Besides the shell, only a few other programs needed to be interactive, such as editors and mailers. In contrast, the personal workstations used today, with their raster displays and mice, encourage a different system structure where a large number of programs are interactive and the most common style of interaction is to manipulate individual applications directly with a mouse. Furthermore, the large displays available today make it possible for many interactive applications to be active at once, whereas this was not practical with the smaller screens of ten years ago.

Unfortunately, few of today's interactive applications have the power of the shell or Emacs command languages. Where good command languages exist, they tend to be tied to specific programs. Each new interactive application requires a new command language to be developed. In most cases application programmers do not have the time or inclination to implement a general-purpose facility (particularly if the application itself is simple), so the resulting command languages tend to have insufficient power and clumsy syntax.

Tcl is an application-independent command language. It exists as a C library package that can be used in many different programs. The Tcl library provides a parser for a simple but fully programmable command language. The library also implements a collection of built-in commands that provide general-purpose programming constructs such as variables, lists, expressions, conditionals, looping, and procedures. Individual application programs extend the basic Tcl language with application-specific commands. The Tcl library also provides a set of utility routines to simplify the implementation of tool-specific commands.

I believe that Tcl is particularly useful in a windowing environment, and that it provides two advantages. First, it can be used as a general-purpose mechanism for programming the interfaces of applications. If a tool is based on Tcl, then it should be relatively easy to modify the application's user interface and to extend the interface with new commands. Second, and more important, Tcl provides a uniform framework for communication between tools. If used uniformly in all tools, Tcl will make it possible for tools to work together more gracefully than is possible today.

The rest of this paper is organized as follows. Section 2 describes the Tcl language as seen by users. Section 3 discusses how Tcl is used in applications, including the C-language interface between application programs and the Tcl library. Section 4 describes how Tcl can be used in a windowing environment to customize interface actions and appearances. Section 5 shows how Tcl can be used as a vehicle for communication between applications, and why this is important. Section 6 presents the status of the Tcl implementation and some preliminary performance measurements. Section 7 compares Tcl to Lisp, Emacs, and NeWS, and Section 8 concludes the paper.

2. The Tcl Language

In a sense, the syntax of the Tcl language is unimportant: any programming language, whether it is C [6], Forth [4], Lisp [1], or Postscript [2], could provide many of the same programmability and communication advantages as Tcl. This suggests that the best implementation approach is to borrow an existing language and concentrate on providing a convenient framework for the use of that language. However, the environment for an embeddable command language presents an unusual set of constraints on the language, which are described below. I eventually decided that a new language designed from scratch could probably meet the constraints with less implementation effort than any existing language.

Tcl is unusual because it presents two different interfaces: a textual interface to users who issue Tcl commands, and a procedural interface to the applications in which it is embedded. Each of these interfaces must be simple, powerful, and efficient. There were four major factors in the language design:

[1] **The language is for commands.** Almost all Tcl ‘‘programs’’ will be short, many only one line long. Most programs will be typed in, executed once or perhaps a few times, and then discarded. This suggests that the language should have a simple syntax so that it is easy to type commands. Most existing programming languages have complex syntax; the syntax is helpful when writing long programs but would be clumsy if used for a command language.

[2] **The language must be programmable.** It should contain general programming constructs such as variables, procedures, conditionals, and loops, so that users can extend the built-in command set by writing Tcl procedures. Extensibility also argues for a simple syntax: this makes it easier for Tcl programs to generate other Tcl programs.

[3] **The language must permit a simple and efficient interpreter.** For the Tcl library to be included in many small programs, particularly on machines without shared-library facilities, the interpreter must not occupy much memory. The mechanism for interpreting Tcl commands must be fast enough to be usable for events that occur hundreds of times a second, such as mouse motion.

[4] **The language must permit a simple interface to C applications.** It must be easy for C applications to invoke the interpreter and easy for them to extend the built-in commands with application-specific commands. This factor was one of the reasons why I decided not to use Lisp as the command language: Lisp’s basic data types and storage management mechanisms are so different than those of C that it would be difficult to build a clean and simple interface between them. For Tcl I used a data type (string) that is natural to C.

2.1. Tcl Language Syntax

Tcl’s basic syntax is similar to that of the UNIX shells: a command consists of one or more fields separated spaces or tabs. The first field is the name of a command, which may be either a built-in command, an application-specific command, or a procedure consisting of a sequence of Tcl commands. Fields after the first one are passed to the command as arguments. Newline characters are used as command separators, just as in the UNIX shells, and semi-colons may be used to separate commands on the same line. Unlike the UNIX shells, each Tcl command returns a string result, or the empty string if a return value isn’t appropriate.

There are four additional syntactic constructs in Tcl, which give the language a Lisp-like flavor. Curly braces are used to group complex arguments; they act as nestable quote characters. If the first character of an argument is a open brace, then the argument is not terminated by white space. Instead, it is terminated by the matching close brace. The argument passed to the command consists of everything between the braces, with the enclosing braces stripped off. For example, the command

```
set a {dog cat {horse cow mule} bear}
```

will receive two arguments: ``a'' and ``dog cat {horse cow mule} bear''. This particular command will set the variable a to a string equal to the second argument. If an argument is enclosed in braces, then none of the other substitutions described below is made on the argument. One of the most common uses of braces is to specify a Tcl subprogram as an argument to a Tcl command.

The second syntactic construct in Tcl is square brackets, which are used to invoke command substitution. If an open bracket appears in an argument, then everything from the open bracket up to the matching close bracket is treated as a command and executed recursively by the Tcl interpreter. The result of the command is then substituted into the argument in place of the bracketed string. For example, consider the command

```
set a [format {Santa Claus is %s years old} 99]
```

The format command does printf-like formatting and returns the string ``Santa Claus is 99 years old'', which is then passed to set and assigned to variable a.

The third syntactic construct is the dollar sign, which is used for variable substitution. If it appears in an argument then the following characters are treated as a variable name; the contents of the variable are substituted into the argument in place of the dollar sign and name. For example, the commands

```
set b 99
set a [format {Santa Claus is %s years old} $b]
```

result in the same final value for a as the single command in the previous paragraph. Variable substitution isn't strictly necessary since there are other ways to achieve the same effect, but it reduces typing.

The last syntactic construct is the backslash character, which may be used to insert special characters into arguments, such as curly braces or non-printing characters.

2.2. Data Types

There is only one type of data in Tcl: strings. All commands, arguments to commands, results returned by commands, and variable values are ASCII strings. The use of strings throughout Tcl makes it easy to pass information back and forth between Tcl library procedures and C code in the enclosing application. It also makes it easier to pass Tcl-related information back and forth between machines of different types.

Although everything in Tcl is a string, many commands expect their string arguments to have particular formats. There are three particularly common formats for strings: lists, expressions, and commands. A list is just a string containing one or more fields separated by white space, similar to a command. Curly braces may be used to enclose complex list elements; these complex list elements are often lists in their own right, as in Lisp. For example, the string

```
dog cat {horse cow mule} bear
```

is a list with four elements, the third of which is a list with three elements. Tcl provides commands for a number of list-manipulation operations, such as creating lists, extracting elements, and computing list lengths.

The second common form for a string is a numeric expression. Tcl expressions have the same operators and precedence as expressions in C. The expr Tcl command evaluates a string as an expression and returns the result (as a string, of course). For example, the command

```
expr {($a < $b) || ($c != 0)}
```

returns ``1'' if the numeric value of variable a is less than that of variable b, or if variable c is zero; otherwise it returns ‘‘`0`’’. Several other commands, such as `if` and `for`, expect one or more of their arguments to be expressions.

The third common interpretation of strings is as commands (or sequences of commands). Arguments of this form are used in Tcl commands that implement control structures. For example, consider the following command:

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

The `if` command receives two arguments here, each of which is delimited by curly braces. `If` is a built-in command that evaluates its first argument as an expression; if the result is non-zero, `if` executes its second argument as a Tcl command. This particular command swaps the values of the variables `a` and `b` if `a` is less than `b`.

Tcl also allows users to define command procedures written in the Tcl language. I will refer to these procedures as *tclproc*’s, in order to distinguish them from other procedures written in C. The `proc` built-in command is used to create a tclproc. For example, here is a Tcl command that defines a recursive factorial procedure:

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

The `proc` command takes three arguments: a name for the new tclproc, a list of variable names (in this case the list has only a single element, `x`), and a Tcl command that comprises the body of the tclproc. Once this `proc` command has been executed, `fac` may be invoked just like any other Tcl command. For example

```
fac 4
```

will return the string ‘‘`24`’’.

Figure 1 lists all of the built-in Tcl commands in groups. In addition to the commands already mentioned, Tcl provides commands for manipulating strings (comparison, matching, and `printf/scanf`-like operations), commands for manipulating files and file names, and a command to fork a subprocess and return the subprocess’s standard output as result. The built-in Tcl commands provide a simple but complete programming language. The built-in facilities may be extended in three ways: by writing tclprocs; by invoking other programs as subprocesses; or by defining new commands with C procedures as described in the next section.

3. Embedding Tcl in Applications

Although the built-in Tcl commands could conceivably be used as a stand-alone programming system, Tcl is really intended to be embedded in application programs. I have built several application programs using Tcl, one of which is a mouse-based editor for X called *mx*. In the rest of the paper I will use examples from *mx* to illustrate how Tcl interacts with its enclosing application.

An application using Tcl extends the built-in commands with a few additional commands related to that particular application. For example, a clock program might provide additional commands to control how the clock is displayed and to set alarms; the *mx* editor provides additional commands to read a file from disk, display it in a window, select and modify ranges of

Control
`break, case, continue, eval, for, foreach, if`

Variables and Procedures
`global, proc, return, set`

List Manipulation
`concat, index, length, list, range`

Expressions
`expr`

String Manipulation
`format, scan, string`

File Manipulation
`file, glob, print, source`

Invoking Subprocesses
`exec`

Miscellaneous
`catch, error, info, time`

Figure 1. The built-in Tcl commands. This set of commands is available to any application that uses Tcl. Additional commands may be defined by the application.

bytes, and write the modified file back to disk. An application programmer need only write the application-specific commands; the built-in commands provide programmability and extensibility ‘‘for free’’. To users, the application-specific commands appear the same as the built-in commands.

Figure 2 shows the relationship between Tcl and the rest of an application. Tcl is a C library package that is linked with the application. The Tcl library includes a parser for the Tcl language, procedures to execute the built-in commands, and a set of utility procedures for things like expression evaluation and list management. The parser includes an extension interface that may be used to extend the language’s command set.

To use Tcl, an application first creates an object called an *interpreter*, using the following library procedure:

```
Tcl_Interp * Tcl_CreateInterp()
```

An interpreter consists of a set of commands, a set of variable bindings, and a command execution state. It is the basic unit manipulated by most of the Tcl library procedures. Simple applications will use only a single interpreter, while more complex applications may use multiple interpreters for different purposes. For example, *mx* uses one interpreter for each window on the screen.

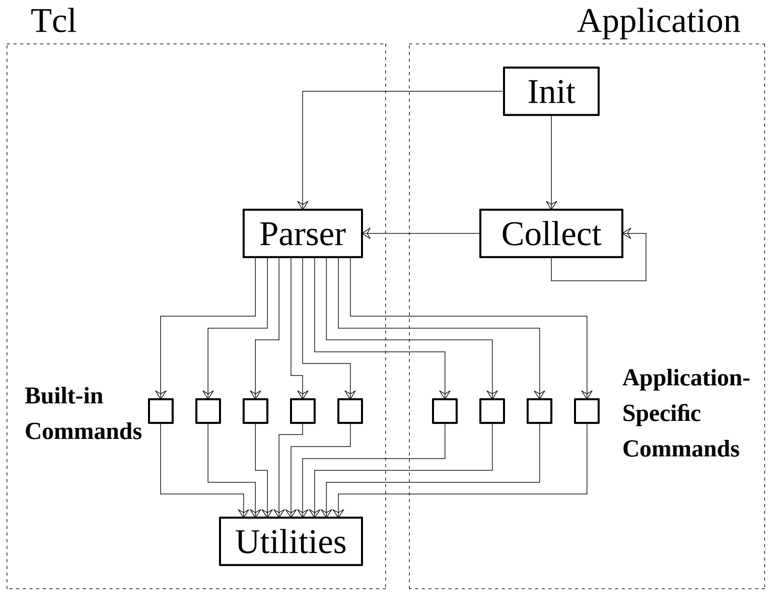

Figure 2. The Tcl library provides a parser for the Tcl language, a set of built-in commands, and several utility procedures. The application provides application-specific commands plus procedures to collect commands for execution. The commands are parsed by Tcl and then passed to relevant command procedures (either in Tcl or in the application) for execution.

Once an application has created an interpreter, it calls the `Tcl_CreateCommand` procedure to extend the interpreter with application-specific commands:

```
typedef int (*Tcl_CmdProc)(ClientData clientData,
     Tcl_Interp *interp, int argc, char *argv[]);

Tcl_CreateCommand(Tcl_Interp *interp, char *name,
     Tcl_CmdProc proc, ClientData clientData)
```

Each call to `Tcl_CreateCommand` associates a particular command name (`name`) with a procedure that implements that command (`proc`) and an arbitrary single-word value to pass to that procedure (`clientData`).

After creating application-specific commands, the application enters a main loop that collects commands and passes them to the `Tcl_Eval` procedure for execution:

```
int Tcl_Eval(Tcl_Interp *interp, char *cmd)
```

In the simplest form, an application might simply read commands from the terminal or from a file. In the *mx* editor Tcl commands are associated with events such as keystrokes, mouse buttons, or menu activations; each time an event occurs, the corresponding Tcl command is passed to `Tcl_Eval`.

The `Tcl_Eval` procedure parses its `cmd` argument into fields, looks up the command name in the table of those associated with the interpreter, and invokes the command procedure associated with that command. All command procedures, whether built-in or application-specific, are called in the same way, as described in the `typedef` for `Tcl_CmdProc` above. A command procedure is passed an array of strings describing the command's arguments (`argc` and `argv`) plus the `clientData` value that was associated with the command when it was created. `ClientData` is typically a pointer to an application-specific structure containing information needed to execute the command. For example, in *mx* the `clientData` argument points to a per-window data structure describing the file being edited and the window it is displayed in.

Control mechanisms like `if` and `for` are implemented with recursive calls to `Tcl_Eval`. For example, the command procedure for the `if` command evaluates its first argument as an expression; if the result is non-zero, then it calls `Tcl_Eval` recursively to execute its second argument as a Tcl command. During the execution of that command, `Tcl_Eval` may be called recursively again, and so on. `Tcl_Eval` also calls itself recursively to execute bracketed commands that appear in arguments.

Even tclprocs such as `fac` use this same basic mechanism. When the `proc` command is invoked to create `fac`, the `proc` command procedure creates a new command by calling `Tcl_CreateCommand` as illustrated in Figure 3. The new command has the name `fac`. Its command procedure (`proc` in the call to `Tcl_CreateCommand`) is a special Tcl library procedure called `InterpProc`, and its `clientData` is a pointer to a structure describing the tclproc. This structure contains, among other things, a copy of the body of the tclproc (the third argument to the `proc` command). When the `fac` command is invoked, `Tcl_Eval` calls `InterpProc`, which in turn calls `Tcl_Eval` to execute the body of the tclproc. There is some additional code required to associate the argument of the `fac` command (which is passed to `InterpProc` in its `argv` array) with the `x` variable used inside `fac`'s body, and to support variables with local scope, but much of the mechanism for tclprocs is the same as that for any other Tcl command.

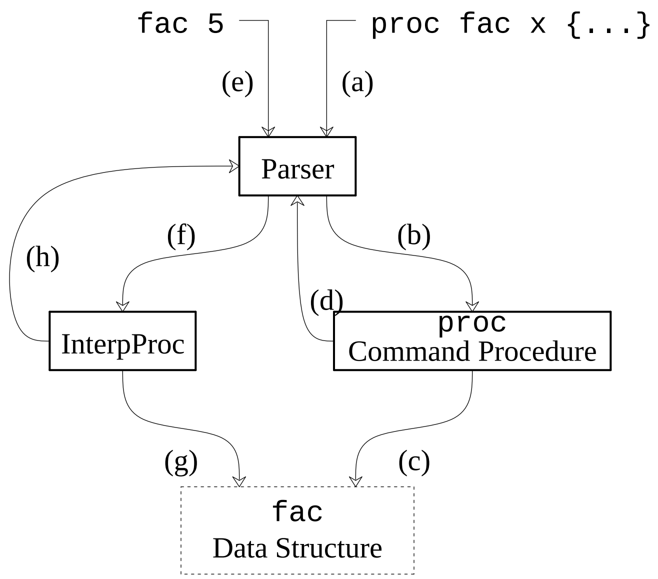

Figure 3. The creation and execution of a tclproc (a procedure written in Tcl): (a) the `proc` command is invoked, e.g. to create the `fac` procedure; (b) the Tcl parser invokes the command procedure associated with `proc`; (c) the `proc` command procedure creates a data structure to hold the Tcl command that is `fac`'s body; (d) `fac` is registered as a new Tcl command, with `InterpProc` as its command procedure; (e) `fac` is invoked as a Tcl command; (f) the Tcl parser invokes `InterpProc` as the command procedure for `fac`; (g) `InterpProc` retrieves the body of `fac` from the data structure; and (h) the Tcl commands in `fac`'s body are passed back to the Tcl parser for execution.

A Tcl command procedure returns two results to `Tcl_Eval`: an integer return code and a string. The return code is returned as the procedure's result, and the string is stored in the interpreter, from which it can be retrieved later. `Tcl_Eval` returns the same code and string to its caller. Table I summarizes the return codes and strings. Normally the return code is `TCL_OK` and the string contains the result of the command. If an error occurs in executing a command, then the return code will be `TCL_ERROR` and the string will describe the error condition. When `TCL_ERROR` is returned (or any value other than `TCL_OK`), the normal action is for nested command procedures to return the same code and string to their callers, unwinding all pending command executions until eventually the return code and string are returned by the top-level call to `Tcl_Eval`. At this point the application will normally display the error message for the user by printing it on the terminal or displaying it in a notifier window.

Return codes other than `TCL_OK` or `TCL_ERROR` cause partial unwinding. For example, the `break` command returns a `TCL_BREAK` code. This causes nested command executions to be unwound until a nested `for` or `foreach` command is reached. When a `for` or `foreach` command invokes `Tcl_Eval` recursively, it checks specially for the `TCL_BREAK` result. When this occurs the `for` or `foreach` command terminates the loop, but it doesn't return the `TCL_BREAK` code to its caller. Instead it returns `TCL_OK`. Thus no higher levels of execution are aborted. The `TCL_CONTINUE` return code is also handled by the `for` and `foreach` commands (they go on to the next loop iteration) and `TCL_RETURN` is handled by the `InterpProc` procedure. Only a few command procedures, like `break` and `for`, know anything about special return codes such as `TCL_BREAK`; other command procedures simply abort whenever they see any return code other than `TCL_OK`.

The `catch` command may be used to prevent complete unwinding on `TCL_ERROR` returns. `Catch` takes an argument that is a Tcl command to execute. It passes the command to `Tcl_Eval` for execution, but always returns `TCL_OK`. If an error occurs in the command, `catch`'s command procedure detects the `TCL_ERROR` return value from `Tcl_Eval`, saves information about the error in Tcl variables, and then returns TCL_OK to its caller. In almost all cases I think the best response to an error is to abort all command invocations and notify the user; `catch` is provided for those few occasions where an error is expected and can be handled without aborting.

4. Tcl and Window Applications

An embeddable command language like Tcl offers particular advantages in a windowing environment. This is partly because there are many interactive programs in a windowing environment (hence many places to use a command language) and partly because configurability

Return Code	Meaning	String
`TCL_OK`	Command completed normally	Result
`TCL_ERROR`	Error occurred in command	Error message
`TCL_BREAK`	Should abort innermost loop	None
`TCL_CONTINUE`	Should skip innermost iteration	None
`TCL_RETURN`	Should return from innermost procedure	Procedure result

Table I. Each Tcl command returns a code describing what happened and a string that provides additional information. If the return code is not `TCL_OK`, then nested command executions unwind and return the same code, until reaching top-level or some command that is prepared to deal with the exceptional return code.

is important in today's windowing environments and a language like Tcl provides the flexibility to reconfigure. Tcl can be used for two purposes in a window application: to configure the application's interface *actions*, and to configure the application's interface *appearance*. These two purposes are discussed in the paragraphs below.

The first use of Tcl is for interface actions. Ideally, each event that has any importance to the application should be bound to a Tcl command. Each keystroke, each mouse motion or mouse button press (or release), and each menu entry should be associated with a Tcl command. When the event occurs, it is first mapped to its Tcl command and then executed by passing the command to `Tcl_Eval`. The application should not take any actions directly; all actions should first pass through Tcl. Furthermore, the application should provide Tcl commands that allow the user to change the Tcl command associated with any event.

In interactive windowing applications, the use of Tcl will probably not be visible to beginning users: they will manipulate the applications using buttons, menus, and other interface components. However, if Tcl is used as an intermediary for all interface actions then two advantages accrue. First, it becomes possible to write Tcl programs to reconfigure the interface. For example, users will be able to rebind keystrokes, change mouse buttons, or replace an existing operation with a more complex one specified as a set of Tcl commands or tclprocs. The second advantage is that this approach forces all of the application's functionality to be accessible through Tcl: anything that can be invoked with the mouse or keyboard can also be invoked with Tcl programs. This makes it possible to write tclprocs that simulate the actions of the program, or that compose the program's basic actions into more powerful actions. It also permits interactive sessions to be recorded and replayed as a sequence of Tcl commands (see Section 5).

The second use for Tcl in a window application is to configure the appearance of the application. All of the application's interface components (''widgets'' in X terminology), such as labels, buttons, text entries, menus, and scrollbars, should be configured using Tcl commands. For example, in the case of a button the application (or the button widget code) should provide Tcl commands to change the button's size and location, its text, its colors, and the action (a Tcl command, of course) to invoke when the button is activated. This makes it possible for users to write Tcl programs to personalize the layout and appearance of the applications they use. The most common use of such reconfigurability would probably be in Tcl command files read by programs automatically when they start execution. However, the Tcl commands could also be used to change an application's appearance while it is running, if that should prove useful.

If Tcl is used as described above, then it could serve as a specification language for user interfaces. User interface editors could be written to display widgets and let users re-arrange them and configure attributes such as colors and associated Tcl commands. The interface editor could then output information about the interface as a Tcl command file to be read by the application when it starts up. Some current interface editors output C code which must then be compiled into the application [7]; unfortunately this approach requires an application to be recompiled in order to change its interface (or, alternatively, it requires a dynamic-code-loading facility). If Tcl were used as the interface specification language then no recompilation would be necessary and a single application binary could support many different interfaces.

5. Communication Between Applications

The advantages of an embedded command language like Tcl become even greater if all of the tools in an environment are based on the same language. First, users need only learn one basic command language; to move from one application to another they need only learn the (few?) application-specific commands for the new application. Second, generic interface editors become possible, as described in the previous section. Third, and most important in my view, Tcl can provide a means of communication between applications.

I have implemented a communication mechanism for X11 in the form of an additional Tcl command called `send`. For `send` to work, each Tcl interpreter associated with an X11 application is given a textual name, such as `xmh` for an X mail handler or `mx.foo.c` for a window in which *mx* is displaying a file named `foo.c`. The `send` command takes two arguments: the name of an interpreter and a Tcl command to execute in that interpreter. `Send` arranges for the command to be passed to the process containing the named interpreter; the command is executed by that interpreter and the results (return code and string) are returned to the application that issued the `send` command.

The X11 implementation of `send` uses a special property attached to the root window. The property stores the names of all the interpreters plus a window identifier for each interpreter. A command is sent to an interpreter by appending it to a particular property in the interpreter's associated window. The property change is detected by the process that owns the interpreter; it reads the property, executes the command, and appends result information onto a property associated with the sending application. Finally, the sending application detects this change of property, reads the result information, and returns it as the result of the `send` command.

The `send` command provides a powerful way for one application to control another. For example, a debugger could send commands to an editor to highlight the current source line as it single-steps through a program. Or, a user interface editor could use `send` to manipulate an application's interface directly: rather than modifying a dummy version of the application's interface displayed by the interface editor, the interface editor could use `send` to modify the interface of a ''live'' application, while also saving the configuration for a configuration file. This would allow an interface designer to try out the look and feel of a new interface incrementally as changes are made to the interface.

Another example of using `send` is for changing user preferences. If one user walks up to a display that has been configured for some other user, the new user could run a program that finds out about all the existing applications on the screen (by querying the property that contains their names), reads the new user's configuration file for each application, and sends commands to that application to reconfigure it for the new user's preferences. When the old user returns, he or she could invoke the same program to restore the original preferences.

`Send` could also be used to record interactive sessions involving multiple applications and then replay the sessions later (e.g. for demonstration purposes). This would require an additional Tcl command called `trace`; `trace` would take a single argument (a Tcl command string) and cause that command string to be executed before each other command was executed in that interpreter. Within a single application, `trace` could be used to record each Tcl command before it is executed, so that the commands could be replayed later. In a multi-application environment, a recorder program could be built using `send`. The recorder sends a `trace` command to each application to be recorded. The `trace` command arranges for information to be sent back to the recorder about each command executed in that application. The recorder then logs information about which applications executed which commands. The recorder can re-execute the commands by `send`-ing them back to the applications again. The `trace` command does not yet exist in Tcl, but it could easily be added.

`Send` provides a much more powerful mechanism for communication between applications than is available today. The only easy-to-use form of communication for today's applications is the selection or cut buffer: a single string of text that may be set by one application and read by another. `Send` provides a more general form of communication akin to remote procedure call [3]. If all of an application's functionality is made available through Tcl, as described in Section 4, then `send` makes all of each application's functionality available to other applications as well.

If Tcl (and `send`) were to become widely used in window applications, I believe that a better kind of interactive environment would arise, consisting of a large number of small specialized applications rather than a few monolithic ones. Today's applications cannot communicate with each other very well, so each application must incorporate all the functionality that it needs. For example, some window-based debuggers contain built-in text editors so that they can highlight the current point of execution. With Tcl and `send`, the debugger and the editor could be distinct programs, with each `send`-ing commands to the other as necessary. Ideally, monolithic applications could be replaced by lots of small applications that work together in exciting new ways, just as the UNIX shells allowed lots of small text processing applications to be combined together. I think that Tcl, or some other language like it, will provide the glue that binds together the windowing applications of the 1990's.

6. Status and Performance

The Tcl language was designed in the fall of 1987 and implemented in the winter of 1988. In the spring of 1988 I incorporated Tcl into the *mx* editor (which already existed, but with an inferior command language), and also into a companion terminal emulator called Tx. Both of these programs have been in use by a small user community at Berkeley for the last year and a half. All of the Tcl language facilities exist as described above, except that the `send` command is still in prototype form and `trace` hasn't been implemented. Some of the features described in Section 4, such as menu and keystroke bindings, are implemented in *mx*, but in an *ad hoc* fashion: Tcl is not yet integrated with a widget set. I am currently building a new toolkit and widget set that is based entirely on Tcl. When it is completed, I expect it to provide all of the features described in Section 4. As of this writing, the implementation has barely begun.

Table II shows how long it takes Tcl to execute various commands on two different workstations. On Sun-3 workstations, the average time for simple commands is about 500 microseconds, while on DECstation 3100's the average time per command is about 160 microseconds. Although *mx* does not currently use a Tcl command for each mouse motion event, the times in Table II suggest that this would be possible, even on Sun-3 workstations, without significant degradation of response. For example, if mouse motion events occur 100 times per second, the Tcl overhead for dispatching one command per event will consume only about 1-2% of a Sun-3 processor. For the ways in which Tcl is currently used (keystroke and menu bindings consisting of a few commands), there are no noticeable delays associated with Tcl. For application-specific commands such as those for the *mx* editor, the time to execute the command is much greater than the time required by Tcl to parse it and call the command procedure.

The Tcl library is small enough to be used in a wide variety of programs, even on systems without mechanisms for sharing libraries. The Tcl code consists of about 7000 lines of C code (about half of which is comments). When compiled for a Motorola 68000, it generates about 27000 bytes of object code.

7. Comparisons

The Tcl language has quite a bit of surface similarity to Lisp, except that Tcl uses curly braces or brackets instead of parentheses and no braces are needed around the outermost level of a command. The greatest difference between Tcl and Lisp is that Lisp evaluates arguments by default, whereas in Tcl arguments are not evaluated unless surrounded by brackets. This means that more typing effort is required in Tcl if an argument is to be evaluated, and more typing effort is required in Lisp if an argument is to be quoted (not evaluated). It appeared to me that no-evaluation is usually the desired result in arguments to a command language, so I made this the

Tcl Command	Sun-3 Time (microseconds)	DS3100 Time (microseconds)
`set a 1`	225	57
`list abc def ghi jkl`	460	138
`if {4 > 3} {set a 1}`	700	220

`proc fac x {`
`    if {$x == 1} {return 1}`
`    return [expr {$x*[fac [expr $x-1]]}]`
`}`
	1280	380

`fac 5`	11250	3630

Table II. The cost of various Tcl commands, measured on a Sun-3/75 workstation and on a DECstation 3100. The command `fac 5` executes a total of 23 Tcl commands, for an average command time of about 500 microseconds on a Sun-3 or 160 microseconds on a DECstation 3100.

default in Tcl. Tcl also has fewer data types than Lisp; this was done in order to simplify the interface between the Tcl library and an enclosing C application.

The Emacs editor is similar to Tcl in that it provides a framework that can be used to control many different application programs. For example, subprocesses can be run in Emacs windows and users can write Emacs command scripts that (a) generate command sequences for input to the applications and (b) re-format the output of applications. This allows users to embellish the basic facilities of applications, edit their output, and so on. The difference between Emacs and Tcl is that the programmability is centralized in Emacs: applications cannot talk to each other unless Emacs acts as intermediary (e.g. to set up a new communication mechanism between two applications, code must be written in Emacs to pass information back and forth between the applications). The Tcl approach is decentralized: each application has its own command interpreter and applications may communicate directly with each other.

Lastly, it is interesting to compare Tcl to NeWS [9], a window system that is based on the Postscript language. NeWS allows applications to down-load Postscript programs into the window server in order to change the user interface and modify other aspects of the system. In a sense, this is similar to the `send` command in Tcl, in that applications may send programs to the server for execution. However, the NeWS mechanism is less general than Tcl: NeWS applications generate Postscript programs as output but they do not necessarily respond to Postscript programs as input. In other words, NeWS applications can affect each others' interfaces, by controlling the server, but they cannot directly invoke each others' application-specific operations as they can with Tcl.

To summarize, the Tcl approach is less centralized than either the Emacs or NeWS approaches. For a windowing environment with large numbers of independent tools, I think the decentralized approach makes sense. In fairness to Emacs, it's important to point out that Emacs wasn't designed for this environment, and that Emacs works quite nicely in the environment for which it was designed (ASCII terminals with batch-style applications). It's also worth noting that direct communication between applications was not an explicit goal of the NeWS system design.

8. Conclusions

I think that Tcl could improve our interactive environments in three general ways. First, Tcl can be used to improve individual tools by providing them with a programmable command language; this allows users to customize tools and extend their functionality. Second, Tcl can provide a uniform command language across a range of tools; this makes it easier for users to program the tools and also allows tool-independent facilities to be built, such as interface editors. Third, Tcl provides a mechanism for tools to control each other; this encourages a more modular approach to windowing applications and makes it possible to re-use old applications in new ways. In my opinion the third benefit is potentially the most important.

My experiences with Tcl so far are positive but limited. Tcl needs a larger user community and a more complete integration into a windowing toolkit before it can be fully evaluated. The Tcl library source code is currently available to the public in a free, unlicensed form, and I hope to produce a Tcl-based toolkit in the near future.

9. Acknowledgments

The members of the Sprite project acted as guinea pigs for the editor and terminal emulator based on Tcl; without their help the language would not have evolved to its current state. Fred Douglis, John Hartman, Ken Shirriff, and Brent Welch provided helpful comments that improved the presentation of this paper.

10. References

[1] Abelson, H. and Sussman, G.J. *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

[2] Adobe Systems, Inc. *Postscript Language Tutorial and Cookbook*, Addison-Wesley, Reading, MA, 1985.

[3] Birrell, A. and Nelson, B. ‘‘Implementing Remote Procedure Calls.’’ *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1986, pp. 39-59.

[4] Brodie, L. *Starting FORTH: An Introduction to the FORTH Language and Operating System for Beginners and Professionals*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[5] Kernighan, B.W. and Pike, R. *The UNIX Programming Environment*, Prentice Hall, Englewood Cliffs, NJ, 1984.

[6] Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*, Second Edition, Prentice Hall, Englewood Cliffs, NJ, 1988.

[7] Mackey, K., Downs, M., Duffy, J., and Leege, J. ‘‘An Interactive Interface Builder for Use with Ada Programs,’’ *Xhibition Conference Proceedings*, 1989.

[8] Stallman, R. *GNU Emacs Manual*, Fourth Edition, Version 17, February 1986.

[9] Sun Microsystems, Inc. *NeWS Technical Overview*, Sun Microsystems, Inc. PN 800-1498-05, 1987.