The following paper was originally published in the
Proceedings of the Fifth Annual Tcl/Tk Workshop
Boston, Massachusetts, July 1997

TclOSAScript - Exec for MacTcl

Jim Ingham
Lucent Technologies (now at Sun Microsystems)
Raymond Johnson
Sun Microsystems

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

TclOSAScript - Exec for MacTcl

Jim Ingham
Lucent Technologies
(now at Sun Microsystems)
jim.ingham@eng.sun.com

Raymond Johnson
Sun Microsystems
rjohnson@eng.sun.com

Abstract:

We describe the TclOSAScript extension to MacTcl. TclOSAScript provides the ability for MacTcl scripts to run scripts in any other OSA compatible language on the Macintosh. Since the OSA is the standard mechanism for interapplication communication on the Mac, this allows MacTcl to run other applications, and provides an exec like facility (though arguably using a much richer communication model.)

I) An Introduction to the Open Scripting Architecture

The usual interapplication communication mechanism for Unix based tools relies on the simple expedient of connecting the standard in and standard out channels of the spawned process to channels of the parent process. The communication between the parent and child processes then mimics the user interaction with the process, namely typing commands on the command line. The "exec" command in Tcl, as well as the open-with-pipe command, use this mechanism to give Tcl access to processes in the surrounding system.

This method, unfortunately, is inapplicable on Macintosh systems. Macintosh applications are only implemented with a Graphical User Interface (GUI). There is no command line, and thus no concept of standard in and standard out. However, this does not mean there is no interapplication communication mechanism. Instead, a much richer method of scripting external tasks is provided by the Open Scripting Architecture (OSA)[1,2].

This problem, and the OSA solution presented here, were first mentioned in papers in the Usenix Tcl/Tk 95 and 96 conferences [3,4]. There are two implementations of this solution that have been presented. Ted Beldung wrote a extension called ASTcl[5] which only worked with AppleScript, and did not use many of the advanced features the OSA offers. TclOSAScript, which was developed concurrently, is a more complete implementation. This is the one we will detail in this paper.

There is a real problem that the OSA aims to solve as well – beyond the fact that there may well not be a text based command interface to tasks in a modern GUI operating system. For while the command-based form of communication is easy to implement, to actually drive an application you need to know the particular language that the application uses, be it simple command line switches, or a real language such as Tcl. There is no generic way to expose the functional elements of an application without tying them to a particular scripting language.

The solution is to define the basic objects and verbs that the application supports in a language neutral way. Then the scripting language must provide a mechanism for querying out these elements. Finally, the operating system will provide a messag-

ing system that will allow any scripting language to drive the objects of the target application.

a) Specification of Verbs & Objects – the aete

The specification of objects is achieved with the *Apple Event Terminology Extension*. This is a resource in the resource fork of the application which lists the 'events' to which the application responds, and the classes of objects in the application.

Examples of events that an application might recognize are the '`get`', '`set`' and '`create`' events, which just provide access to the objects in the application. These form part of the "Standard Suite" of commands that all well designed OSA applications should support. There are also more specialized verbs, such as '`select`' or '`revert`' in a Word Processor app or '`download`' and '`view file list`' in the FTP client application Fetch. Events also take parameters, such as which word to select in a word processor, or which file to download in Fetch.

Examples of objects are the windows of an application, or the words and paragraphs in a word processor window. There is also a way of specifying a containment hierarchy, so each object can be contained in other objects, and can contain other objects, . Finally, the objects may take qualifiers. An example of a moderately complicated object specifier, in AppleScript dialect, is: "`the file "mactcltk-full-8.0a2" in the transfer window "ftp.sunlabs.com" of the application "Fetch 3.0.2"`".

The aete makes the connection between the text phrase which specifies an object or verb, and some packed 4 character code which will occur in the compiled form of the interapplication messages. It also specifies the parameters for the verbs (with the notion of required and optional parameters), and the containment hierarchy of the objects. Once you know the aete of an application, you know the form any message to that application must take.

b) The Messaging system – Apple Events

The messaging system in the OSA is governed by the Apple Event Manager. An Apple Event is a data structure that describes an event in terms of the aforementioned packed 4 character codes. It starts with the main verb, and then usually has a direct object with its qualifiers, and perhaps other parameters. Apple Events support nesting, so that a single event can support recursive evaluation (much like the [] syntax in Tcl).

One application under the MacOS sends an Apple Event to another application by building up the Apple Event data structure, and then passing it to the Apple Event Manager, which places the event in the event queue of the target application. The target application then handles this event by parsing up its contents and performing the required task.

c) OSA compliant languages

All interapplication communication is carried out, under the hood, by the exchange of Apple Events. However, this is too low level an exchange mechanism to be useful to the average user.

The job of an OSA compliant language, then, is to wrap a more human syntax around the construction of the Apple Event messages. The approved method for doing this is to read the aete of a target application, and import the verbs and nouns found therein into its own syntax in a natural way. Done this way, the aete can serve as documentation for how to script each language, and the language can dynamically appropriate new applications as they are encountered.

Of the two major OSA compliant languages, only AppleScript[6] dynamically reads the aete of the target application. UserLand Frontier[7] has a static glue table for each application that relates the event and object codes with commands in the Frontier language. This limits the number of scriptable applications that Frontier can actually drive.

d) OSA Components

The final part of the OSA architecture is the OSA component. This is an implementation of the notion that a scripting language is not so much a part of the application, as a service that is provided to applications, to aid them in exposing their operative parts.

Tcl has been very successful precisely because it makes it easy to bind together the functional elements of the application without having to recompile the application. But to make use of Tcl, the application must embed a Tcl interpreter within the object code of the application. This ties the use of the application to Tcl; you cannot dynamically choose which scripting language you want to employ.

The OSA goes further to make the operating system the manager of scripting solutions. The provider of a scripting language writes some glue code that registers the scripting language with the oper-

ating system. Applications request a connection to the scripting component, and then send script data to that component to be evaluated. The script data can even contain tags identifying its language, so that the application can receive scripting data in any of the available languages, pass it to the OSA component manager, which will in turn route it to the appropriate component.

This connection can be used in two ways. One is to allow an application to drive other applications, so that it does not have to duplicate their functionality. A good example of this is the MicroSoft Internet Explorer, which hands off e-mail, FTP and NNTP requests to the user's favorite e-mail, FTP and Usenet clients using the OSA.

A deeper use of the OSA is to "factor" the user interface elements of the application from its core functionality. The user interface elements do not directly call the subroutines that do the work of the application. Rather they send AppleEvents back to the application, which receives these events, and then dispatches them to the appropriate internal routines.

The advantage of this is that you can then "attach" scripts to the UI elements, which scripts do the work of dispatching the AppleEvents. This immediately provides an architecture much like HyperCard, or the Tcl/Tk callbacks, but with the added benefit that the callback language is left up to the user's discretion.

II) Tcl and the OSA

There are three steps to fit Tcl into the OSA architecture. The first is to allow Tcl to use the services offered by the OSA components installed on the system. This is the function of the TclOSAScript extension which is described in the rest of this paper. The next two steps have not yet been completed. The first of these is to provide a Tcl command to build and dispatch Apple Events. The final step is to install Tcl/Tk as an OSA component in its own right, so that Tcl would be an option in all the major Scripting development environments.

TclOSAScript

The first stage of this incorporation is completed. The TclOSAScript extension allows Tcl scripts to connect to any available OSA component, and send scripts off to be evaluated by that component. Currently connections to both of the popular OSA languages, AppleScript and UserLand Frontier have been tested. TclOSAScript will be included as a shareable library in the 8.0 release of MacTcl. What follows is a general description of the TclOSAScript extension.

1) The OSA command

The first step in using any OSA component is to get a connection to an instance of the component. At startup, TclOSAScript scans the list of OSA components, opens a connection to each one that is found, and creates a Tcl command to access that component. So if the user has AppleScript installed on their machine, an `AppleScript` command will be created. If UserLand Frontier is running at startup, a `UserLand` command will be created, and so on.

There is also a generic command, `OSA`, that will allow other connections to be opened, or closed, and will allow the user to query the list of components. This is useful, for instance, to open connections with other languages.

An example of this use is communication with the UserLand language. The UserLand OSA component is installed only while the Frontier application is running. So if you wanted to use Frontier, you could open it (by using the `AppleScript` command), and then use the `OSA` command to open a connection to the UserLand component...

It is currently not useful to open multiple connections to the same component. Since the calls to the OSA components are synchronous, having several connections to a single OSA component does not gain anything.

2) Compiling and Executing Scripts:

Once you have an open component, you can send scripts to it, to be executed. The simplest way to do this is with the `execute` subcommand (see fragment 1):

```
AppleScript execute {
    tell application "Fetch 3.0.2"
        download (url "ftp://" & ¬
              "ftp.sunlabs.com" & ¬
              "/pub/tcl/mac/"   & ¬
              "mactcltk-full-8.0a2.sea.hqx")
    end tell
}
```

Fragment 1

```
set getNewFiles [AppleScript compile {
    tell application "Fetch 3.0.2"
        open (url "ftp://ftp.sunlabs.com/pub/tcl/mac")
        set thisWin to (transfer window "ftp.sunlabs.com")
        set nfiles to (count remote item in thisWin)
        set retVal to {}
        repeat with i from 1 to nfiles
            if the modification date of (remote item i of thisWin) > refDate then
                download (remote item i of thisWin)
            end if
        end repeat
    end tell
    get retVal
}]
AppleScript execute {set refDate to (date "Saturday, February 1, 1997 3:19:02 PM")}
button .b -text "Check for files" -command "AppleScript run $getNewFiles"
pack .b -padx 6 -pady 6
```

Fragment 2

This will execute a script that uses the Mac application "Fetch" to get the 8.0a2 version of mactcltk from the Sun ftp site...

However, complicated scripts can take some time to compile, so if you want to run the same script over and over, you might want to compile the script once, and then run it many times. For this purpose, there is a `compile`/`run` pair of subcommands.

In the example in Fragment 2, we make a button whose action is to download all the files in the /pub/tcl/mac directory whose date is later than the some reference date (chosen pretty much at random here...):

The "`AppleScript compile`" command passes the script to AppleScript to compile, and returns a script handle for the compiled script. This handle can be passed to the "`AppleScript run`" command, which will execute the script, and return whatever value the script returns.

3) Other Tricks

We have also used another trick of TclOSAScript, to define the variable refDate. AppleScript executes its scripts in "*Script contexts*", which are just namespaces that retain all the variables and procedures that are defined in them. The `AppleScript` command opens a default context which it uses for all script execution. So the "`AppleScript execute`" line in Fragment 2 will set the `refDate` variable which the `getNewFiles` script uses in its execution.

Note that although the notion of a script context is a part of the OSA specification, it is not one of the required parts. So, for instance, UserLand Frontier does not use them. It has its own mechanism for persistence (the object database).

Because Tcl and the code within the AppleScript commands are just strings, we can combine the two languages for even more power. As an example let's extend the above example to always use the current time instead of a set date. The following code uses double quotes to allow Tcl based substitution to occur before the `AppleScript` command compiles the string into AppleScript byte codes.

The example in Fragment 3 will get the current seconds and format the time into a string that is acceptable for the AppleScript `date` command. You must be careful, however, when doing such combinations to make sure you create a valid AppleScript command. A good understanding of Tcl's quoting conventions is required.

```
AppleScript execute "set refDate to (date \"[clock format \
[clock seconds] -format \"%A, %B %d, %Y %X %p\"]\")"
```

Fragment 3

4) Return Values

One other issue is the return values that come back from AppleScript. Suppose that we want to ex-

```
…
set scriptRsrc [AppleScript load $scriptFile]
AppleScript run $scriptRsrc
…
button .f2.view -text "Display Files" -command loadList

proc loadList {} {
       global hostName filePath refDate scriptRsrc
       .f1.lf.lb delete 0 end
       AppleScript execute -variable retList -context $scriptRsrc \
           "listMoreRecent(\"$hostName\",\"$filePath\",date \"$refDate\")"
       eval .f1.lf.lb insert 0 $retList
}

button .f2.load -text "Download" -command {getFiles}

proc getFiles {} {
       global hostName filePath scriptRsrc
       foreach index [.f1.lf.lb curselection] {
              set name [.f1.lf.lb get $index]
           AppleScript execute -context $scriptRsrc \
               "getFiles(\"$hostName\",\"$filePath\",\{\"$name\"\})"
       }
}
```

Fragment 4

pand the following example to query the /pub/tcl/mac directory, and present a list of new files to the user, so that she can choose which ones to download. Then we could write a subroutine in AppleScript that returned a list of new files.

However, this will be returned as an AppleScript formatted list, not a Tcl Formatted list. Now sometimes you may need the AppleScript format (e.g. to pass back to AppleScript). At other times the Tcl format is more appropriate.

In TclOSAScript, the return value of the run command is the AppleScript form of the result. Then we have added a "`-value`" flag to the run and execute commands. You use it to pass the name of a variable to the command, and TclOSAScript will parse AppleScript lists up into Tcl lists, and AppleScript Records into Tcl arrays, and put the result in that variable.

5) Loading Scripts

Finally, Tcl is not the most convenient site for developing AppleScript code. Rather than have to cut and paste from the Script Editor into Tcl, TclOSAScript has a "`load`" command that will load script data from a script resource (which is the format all the AppleScript development environments write out), either in the application itself, or an auxiliary file.

This facility allows for useful methods of user customization for Tcl applications. For instance, a MacTcl application could read the script resources out of all the files in a "Scripts" folder, and populate the "Scripts" menu in the application with them.

Using a stored script, containing two AppleScript subroutines *listMoreRecent* and *getFiles*, we can create the application shown in Figure 1 and Fragment 4, which will display the new files in the given directory, and download the ones chosen in the list. In typical Tcl fashion, the whole application is 60 lines of Tcl, and 24 lines of AppleScript code…

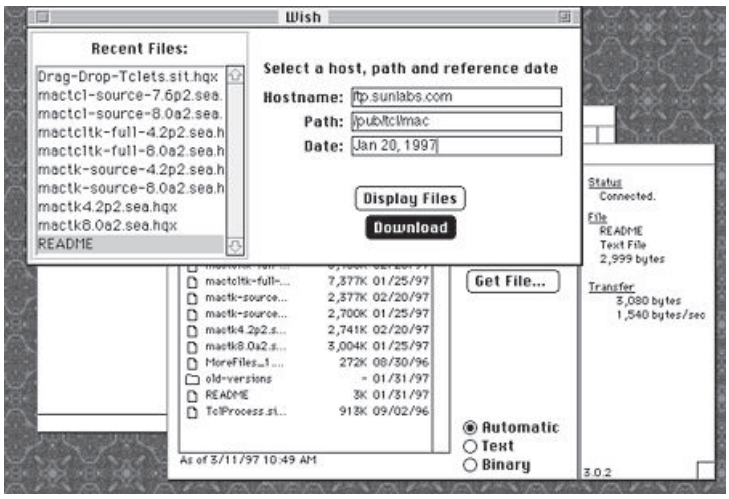

Figure 1

III) Future work

The TclOSAScript extension is almost complete. It would perhaps be useful to provide an asynchronous mode of execution (analogous to `exec` with an `&`) Another major enhancement would be to make each component command run in a separate thread. Then it would be useful to open a connection to, say, the AppleScript component, send it a long-running script. Then open another connection if you have some work to be done before the first task is completed.

Aside from these details, there remain two parts of the complete incorporation of Tcl into the OSA scheme. The first remaining step is to provide a facility to parse the aete of target applications, and a Tcl based mechanism for dispatching Apple Events based on the verbs and objects found therein.

Parsing the aete is relatively straightforward. However, defining a Tcl representation for AppleEvents will require more work. The nesting of objects that is required to specify elements in a target application can require the construction of Apple Events of considerable complexity. For instance, we might want to query out "Every paragraph of the first window of the application "WordPerfect" whose font is "Times Roman" and whose first word is "Foo"". The equivalent Tcl syntax will have to be rich enough to mirror that.

The last step is to provide Tcl as an OSA component. A proof of concept implementation was done by Vince Darley[8], but this was without Tk, and did not address many issues that will be faced by a full implementation.

IV) Summary

The work that we have done so far, the TclOSAScript extension, has given Tcl the ability to take advantage of the other applications in the Macintosh environment. Our example just used the FTP client program Fetch, but there are many other powerful scriptable applications available, including FileMaker Pro, Quark Express, and both Netscape and MSIE, to mention just a few. This richness is now available to the writers of MacTcl applications.

Conversely, the OSA scriptor now has available the Tk GUI toolkit which has proved a great boon in the UNIX world. Many of the work flow applications that have traditionally been written in AppleScript and Frontier can now be augmented with the sort of sophisticated front-end that can so easily be created in Tcl/Tk. With the native look-and-feel of version 8.0, the power of Tcl/Tk will prove as valuable on the Macintosh as it has in UNIX.

References

[1] Apple Computer
Inside Macintosh - Interapplication Communication
Addison Wesley Publishing Corporation, 1993

[2] Dave Mark
Ultimate Mac Programming
IDG Books WorldWide Inc, 1994

[3] Johnson, R. and Stanton S.
"Cross Platform Support in Tk"
In Proc: Usenix Tcl/Tk Workshop, Toronto, Ontario, Canada, 1995.

[4] Ingham, J.
"Tcl/Tk as an OpenDoc Scripting Part"
In Proc: Usenix Tcl/Tk Workshop, Montery, California, 1996.

[5] Ted Belding
<Ted.Belding@umich.edu>
ASTcl.
http://www-personal.engin.umich.edu/~streak/ASTcl-1.0.sea.hqx

[6] Danny Goodman
Danny Goodman's AppleScript Handbook
Second Edition
Random House, 1994

[7] Dave Winer
UserLand Frontier
http://www.scripting.com/frontier/

[8] Vince Darley
OSATcl
http://www.fas.harvard.edu/~darley/Vince-Downloads.html