Dynamic Extension and Control of Xt Applications

Jan Newmarch
Huimin Xu

Abstract

Given an existing application, extensions/modifications may need to be made, either on a permanent basis or on a local basis. Depending on the situation, source code may or may not be available. This paper considers an extension and control mechanism that does not require source code availability. It is done using an extended version of *tclMotif* that makes use of the X11R6 hook object to create a *tcl* command for every widget in the application. This allows a user (or agent of some kind) to extend, modify and control the functionality of the application through its user interface.

Introduction

The required capabilities of an application will depend on the user of that application, the context in which it is used and many other factors. Well-designed systems will allow at least some means of customisation, such as the use of resources in X. However, this should not be limited to appearance but should also include functionality: it should be possible to add or to remove features from an application.

Usually, modifying the behaviour of an application requires access to the source code, and an intimate knowledge of the internal structure of that application. There are a very few systems, such as emacs, which are designed with extension in mind through Lisp programs. Of course, not everyone will write such extensions, but users will pick and choose the extensions that they wish to use.

GUI programs such as those based on X will often expose a large amount of the internal system to manipulation through the user interface. Extension of functionality can often be done by adding new interface elements which make a limited set of calls into the underlying system, or by modifying those user interface elements already in place. Such applications are those where the user interface elements correspond fairly directly to objects within the application. Editors and file managers are obvious examples.

This paper discusses a system that gives access to the user interface side of an application using a scripting language. The application merely needs to make a call to, and have been linked with, a suitable library. If applications are dynamically linked to X, then a suitably modified version of the Xt library could be used to avoid any interference with the application at all.

This allows a user (or agent of some kind) to extend, modify and control the functionality of the application through its user interface.

The structure of this paper is as follows: the next section describes *tclMotif*, how this can be used to build complete applications, and aspects of the underlying implementation. The next section describes *replayXt*, a test system that can be added to existing Xt applications. This is followed by a section on the Hook object of X11R6. It is then described how the three can be used to create a system whereby *tclMotif* objects can be automatically generated for an Xt application, and the application can then be controlled by *tclMotif* code. This is followed by examples. This allows extensions to be made using script files. Attention then turns to remote agents, and both the `send` command and the x-agent extensions are discussed.

tclMotif

Language choice

There are a large number of bindings of programming languages to Xt toolkits such as Motif. These allow a programmer to write an Xt application using their preferred language. The set is summarised in the Motif FAQ [Sall95]. Some bindings, such as *tclMotif* [Newmarch94a], DTKSH [Pendergrast94], Wafe [Neumann93], etc make the task of building an application much easier by using a simple command language instead of more complex C. These can reduce the size of GUI code by upto 90% over C.

The majority of these bindings support a *single language paradigm*, that is, *all* of the programming is done using the chosen language. The value or completeness of the programming environment is then dependant on how well the binding maps into the language, and how much of Xlib and Xt it manages to make available.

There are a number of systems which support a mixed language paradigm in some way or another. Systems such as Xgen [Buehler92] and BS (Button Shell) [de Figueiredo95] allow applications to call arbitrary shell scripts. Wafe allows pipelines to be set up between applications written in *tcl* and programs written in arbitrary languages. Systems such as UIL or UIMS use a specification language for the GUI components and C for the application code.

The Desktop Korn Shell allows dynamic linkage of object modules in arbitrary languages, but finally relies on programs written in the ksh. On the horizon we still see CORBA with its promise of mixed language programming using an Interface Definition Language (IDL).

The most integrated mixed language system appears to be a system using Interviews with an embedded scripting language [Stabile95]. We are really interested in Xt, and in particular, Motif applications though.

tclMotif offers a fairly complete binding to the Motif and Xt libraries and enough of Xlib where appropriate. It is also much simpler to use than C or many other non-scripting systems. *tcl* itself was designed for a mixed-language system, primarily C with *tcl* as command language. The rest of this section discusses *tcl* and how *tclMotif* is built above the Motif libraries.

tcl

tcl [Ousterhout94] is a simple type-free command language designed to be *embeddable* within arbitrary C applications. This is done by declaring a *tcl* interpreter to the application that can then be used to execute *tcl* commands. The language is extensible by being able to define new commands that execute by calling application-specific code. There is a simple interface between application and interpreter that allows the application to examine and modify the interpreter state, whereas the interpreter can only modify application state through the commands defined to it.

The syntax of *tcl* is a cross between the C shell and C, with some cleanups in the syntax as regards command substitution. A simple program to read integers (one per line) from stdin and calculate the minimum is

```
proc max {x y} {
    if {$x < $y}
    return $y
    return $x
}

gets stdin maximum
while {[gets stdin val] != -1} {
    set maximum [max $val $maximum]
}
puts stdout "largest is $maximum"
```

The interface between a *tcl* interpreter and the application is defined by a set of functions including

```
Tcl_CreateInterp()
Tcl_CreateCommand()
Tcl_SetVar()
Tcl_GetVar()
```

```
Tcl_Eval()
```

These allow the application to query interpreter state (`Tcl_GetVar`), set interpreter state (`Tcl_SetVar`), to define application specific commands to *tcl* (`Tcl_CreateCommand`), and to make the interpreter execute *tcl* code (`Tcl_Eval`).

There are a large number of applications which contain *tcl* interpreters. These include the tcl/Tk system, expect, tclMotif, Visual tcl and WAFE.

tclMotif

tclMotif consists of a binding of all the Motif widgets to *tcl*. It adopts an object-oriented approach, in which each widget becomes a *tcl* command. These objects have methods such as `setValues` and `getValues`. Complex objects such as List have many additional methods. Each widget has an object name which is (approximately) its full name from the toplevel widget, with each component separated by '.'. For example, creating a PushButton and setting its label value can be done by

```
xmPushButton .main.button
.main.button setValues \
    -labelString "Hello world"
```

Callbacks can be added to widgets by a method for each callback:

```
.main.button activateCallback {
    puts stdout "I was pushed"
}
```

Callbacks can include `call_data` information by using substitutions: `%click_count` in callback code will have its value substituted.

tclMotif includes other Xt features such as timers, input handlers, etc.

tclMotif comes as a linkable library. It is generally used linked into an application `moat` which sources a file and enters an Xt event loop.

Example tclMotif application

This is example 4.4 from Dan Heller's Motif programming book, done in *tclMotif*

```
# enter a command, execute it. place output in bottom window,
# command in history list
```

```
xtAppInitialize

xmMainWindow .main managed

# menu with one button in one pulldown
xmMenuBar .main.menuBar managed
xmCascadeButton .main.menuBar.file managed -labelString File
xmPulldownMenu .main.filePane
.main.menuBar.file setValues -subMenuId .main.filePane
xmPushButton .main.filePane.quit managed -labelString Quit
.main.filePane.quit activateCallback "exit 0"

xmScrolledText .main.text managed \
    -rows 5 -columns 40 \
    -editable False -editMode Multi_line_edit

xmCommand .main.command managed -promptString "Command:"
.main.command commandEnteredCallback "exec_cmd {%value}"

.main setValues -menuBar .main.menuBar \
    -commandWindow .main.command \
    -workWindow [.main.text parent]

proc exec_cmd {command} {
    set comm_output [eval exec $command]
    .main.text setValues -value $comm_output
}

. realizeWidget
. mainLoop
```

Implementation aspects

There is a standard method for creating applications with embedded *tcl* interpreters, which is employed by `moat`. An application creates a *tcl* interpreter and then defines the commands that will be added to the basic *tcl* language. These commands are associated with C code within the application that will be called when the interpreter needs to execute the *tcl* command.

For example, the *tcl* creation command for the Motif PushButton is `xmPushbutton`. A simple implementation would associate a C function `CreatePushButton()` with this (*tclMotif* actually optimises the sharing of C code among different widgets). As a component of this function, the Motif C widget creation command `XmCreatePushButton()` is called.

tclMotif adopts the Tk object-oriented approach and defines a command with the widget name.

This in turn needs to be associated with code to handle the methods for that type of widget. A very simplified form of the code looks like:

```
CreatePushButton(interp, argv, ...)
{
    char *name = argv[1];
    Widget w = XmCreatePushButton(..., name, ...);

    Tcl_CreateCommand(interp, name, PushButtonCommand, ...);
}

PushButtonCommand(interp, argv, ...)
{
    char *method = argv[1];
    Widget w = ...

    if (strcmp(method, "manageChild") == 0)
        XtManageChild(w);
    ....
}
```

The `moat` interpeter then sources a file, creating Motif and *tcl* objects, and enters an Xt event loop.

Callbacks are defined as *tclMotif* code, and this along with the *tcl* interpreter are passed into a generic callback hander via the `client_data` field

```
CallbackHandler(w, client_data, call_data)
{
    Tm_ClientData *c = (Tm_ClientData *) client_data;
    Tcl_Interp *interp = c->interp;
    char *callback_cmd = c->command;

    Tcl_GlobalEval(interp, callback_cmd);
}
```

replayXt

Overview of replayXt

replayXt [Newmarch94b] is an action-object system also based on *tcl*. This is defined as a library rather than a standalone application because it is only intended to be used as an embedded system.

The purpose of *replayXt* is to able to deliver automated action replays and for testing of the GUI components of the application. It does not use low-level X events, but instead exploits the object nature of Xt widgets.

This system is set up to control an application by using widget actions. A *tcl* command `callActionProc` invokes the Xt function `XtCallActionProc()`. The widget parameter is determined by using `XtNameToWidget()`. A typical action is

```
callActionProc main.btn Arm()
```

Although *replayXt* attempts to use only the widget actions, some widgets require event information. For example, when Btn1 is released after being pressed within a PushButton, the widget examines the event to determine if it occurred inside or outside the widget. Extra arguments may be needed to contruct a suitable event:

```
callActionProc .main.btn Activate() \
    -type ButtonRelease \
    # ensure within the widget \
    -x 0 -y 0
```

Other aspects of *replayXt* are concerned with imposing delays in execution and with getting widget resource values and screen dumps in order that automated testing can be performed.

Implementation

A *tcl* function such as `callActionProc` is defined similar to the functions of *tclMotif*: it makes a call to Xt functions via a wrapper in C.

It is not adequate to just execute a *tcl* script. There needs to be an interlacing of *tcl* commands and Xt events so that both can be processed properly. Otherwise a "sleep" operation or any other operation taking a long time would just freeze the application until the operation was over.

There is a neat way to solve this that uses aspects of both *tcl* and Xt: *tcl* contains a "trace" mechanism that is invoked before any command is executed. By placing an Xt pending-event loop in the trace function, the finest granularity of both is gained.

tcl commands such as "sleep" can then be implemented by timer functions, where any intermediate event processing is handled by the trace function.

There is a need within such a system, being string based, to find a widget from a string. The only feasible way of doing this is to use `XtNameToWidget()`. This is quite a reasonable way, but forced *replayXt* into an action/object paradigm: there was no way in X11R5 of creating a *tcl* command from a widget, because there it was not possible - apart from gross interference in the source code - to spot every widget creation.

Hook object

The Hook object is new to X11R6, and is designed to meet the needs of "agent" programs that need to know about the creation/realisation/etc of widgets. Callbacks can be registered with this Hook object that are invoked on every generic widget operation.

This allows tracking of widget creation and destruction, with a callback function invoked for every operation of this type. This is precisely what is needed by *replayXt* to move it from an action/object approach to the object/action approach of *tclMotif*. In fact, this allows the two systems to be blended.

It also allows *tclMotif* commands to be added to an arbitrary application, allowing the extension methods to work.

Creating tclMotif objects

A callback function is added to the Hook object for widget creation. This function is executed every time a widget is created, with the widget being passed through in the `call_data` filed. The function is actually executed *after* the widget has been created. It is simple to create a *tcl* command for this object, using `Tcl_CreateCommand()`.

```
HookCreateCallback(w, call_data, client_data)
{
    Widget new_w = call_data->widget;
    char *path = TclMotifName(new_w);
    Tcl_Interp *interp = client_data->interp;

    Tcl_CreateCommand(interp, path, ...);
}
```

How this differs from the *tclMotif* widget creation command is in what creates what. In *tclMotif*, the C creation function has to create both the Motif object and the *tcl* object. Using the Hook create callback, the Motif object has already been created, and only the *tcl* object needs to be made (and the *tclMotif* support structures).

The standard *tclMotif* library can be linked into the application and then the full array of *tclMotif* commands is then available to the application. This is really quite simple - barring a few small problems.

Naming problems

In various places, Motif inserts extra widgets into the widget tree. Using the `XmCreate-Scrolled...()` functions or the `XmCreate...Dialog()` functions creates additional widgets, with names such as `..._SW` and `..._popup`. These were intended for simpler programming, but it is not clear that this goal was achieved :-(

tclMotif deals with this by a similar technique to Motif. A call to one of these functions creates a *tclMotif* object with the name expected by the programmer, but also creates another widget with a name that you don't need to know but which is accessible by the `parent` method

```
xmWarningDialog .warning
...
# now make its parent go away
[.warning parent] destroyWidget
```

Given an arbitrary Xt application, we need to create a string pathname for each widget (which must be unique). Apart from reading source code (or specifications :-), the only reliable way of determining pathnames is by using `editres` or by a mechanism that will tell us a name.

The internal *tclMotif* system is not meant to be known because a *tclMotif* programmer has no need of it, so they are not well documented. Additionally, it will be difficult to reliably spot these extra widgets and map their names onto the *tclMotif* names. Consequently, the names as shown by `editres` are used. These names are those that were intended to be hidden by Motif, but which have eventually been documented. If the application is not "editres aware" then a resource can be set so that all widget names are printed as they are created. A "trial" run can then be used to determine the full set of widgets.

Widgets created by *tclMotif* when the Hook callbacks are in place potentially create two *tcl* commands - one when the *tclMotif* C code creates the *tcl* command, and once when the Hook create callbacks execute. To avoid this, only the *tclMotif* commands are created, and the Hook commands are not. This may introduce an inconsistency in naming widgets, where some widgets have a name derived from a *tclMotif* path, others have a name derived from the full widget path. This is probably not serious, just a nuisance.

Ordering problems

Some Motif widgets create their own children. For example, the SelectionBox widget creates a LabelGadget for the Items label, and a ScrolledList for the items themselves. These are created within the widget's own creation procedure, and their creation procedure terminates before that of the parent! This means that the create callback function will be invoked for the child before it is invoked for the parent.

In the case of *tclMotif*, tables are maintained of widget information and much of this information is inherited from the parent. Since this information is not created until the widget's own creation procedure is over, this means that the parent does not have this when the child is created. It is not hard to use the grand-parent's information instead, but this does invert the expected tree-traversal mechanisms expected of Xt.

While this causes an implementation hiccup, it is probably fine for the application programmer -

in cases such as Motif 2.0 SpinBox where such things as resource changes need to be set on child widgets, one can be sure that the children exist when the Hook object createCallback function is executed for the SpinBox.

Naming duplication

Xt does not require widget names to be unique. Many language bindings to Xt make this requirement though. This is often to map a widget name uniquely to an object command. For example, the Java AWT [Java95] library enforces this. *tclMotif* silently overwrites the previous command name. If an application has used duplicate names, then there is no simple way of handling it.

Executing tclMotif code

Granularity

Just before a *tcl* command is executed, any C function registered by the function

```
Tcl_CreateTrace()
```

is executed. A function is registered that calls `XtProcessEvent()` while there are pending events. This gives the finest granularity possible for both *tcl* commands and X events.

Entering the main loop

Most of the widgets of a typical application are created before entrance to the main event processing loop. Entrance to the main loop is therefore a good place to put much of any added *tclMotif* code. A timer of zero is set to execute *tcl* commands as the event loop starts.

Widget creation

Some aspects of an application, such as dialogs, may not be created before the event loop begins. It would be an error to try to use such widgets in a *tclMotif* program before they are created. For such cases, *tclMotif* code can be executed on the creation of each widget, so that it can ensure that needed widgets are in existence before using them. Using this technique too heavily may slow an application down a little, though.

Startup

replayXt allowed a "rules" file to be loaded as the system was registered. Typically this stored procedures into the *tcl* interpreter for later use by other *tcl* commands.

Resources

For processing of local scripts, the following application resources may be set:

```
*scriptLanguage: tclMotif
*script:   <script>
```

The script may "source" a file or be complete in itself. Such a resource system will allow both *tclMotif* scripts and those of other scripting languages (but not simultaneously!).

Multiple execution points

We have observed the need for *tclMotif* code to be executed at a variety of points - when the *tclMotif* system is first registered, when the event loop starts, and on widget creation. Examples of all three uses are given later. The Hook object also allows callbacks to be registered for other potentially interesting cases such as widget management, SetValues, destruction, etc. It is quite possible that an extension writer may wish to attach *tclMotif* code to any of these.

One possibility is to define multiple application resources: a "createScript", a "manageScript", etc. This would lead to a large number of scripts (and resources!). This may make it difficult to maintain logic and even harder to switch to a new program logic - a single script that maintains its own program structure (perhaps "source"ing other files) would be preferable.

To give a single script with multiple execution points, the script needs to be partitioned into multiple parts. This is best done by defining new *tcl* commands, each corresponding to a separate execution point. That is, we define new commands

```
onEventLoop <command>
onCreate <command>
```

The whole script may then be executed by `Tcl_GlobalEval()`.

Ordinary *tcl* code is be executed directly. This corresponds to "source"ing the rules file of *replayXt*.

The `onEventLoop` commands are attached to a timer with timeout of zero, to execute when `XtAppMainLoop()` begins. This corresponds to running the data file of *replayXt*, but with the added ability from *tclMotif* of being able to create and manipulate widgets.

The `onCreate` commands are attached to the Hook object callback `XtNcreateCallback`, and executes within that callback. Note that we already have a callback in this widget that creates the *tclMotif* object. Since the order of multiple callback functions for a callback is undefined, there could be a potential ordering problem if this was implemented within a separate callback function. So it needs to be executed within the same callback function, after the `Tcl_CreateCommand` call.

A single extension may use both an `onEventLoop` and `onCreate` command. Multiple extensions, built independantly of each other, would use multiple commands. Multiple `onEventLoop` commands may be executed, each with their own timer function, without any modifications. To allow for multiple `onCreate` commands to be executed, a list is maintained of the commands for execution within the callback, rather than a single callback function.

There is also a minor tradeoff between *tcl* and C in this: the C code could examine the widget id, or the *tcl* code could do so. To do this in *tcl* would require making the widget name available to the *tcl* code; there is a mechanism within *tclMotif* callbacks for substition of `call_data` structure information. This is just a bit slower, and would also involve protecting `call_data` structure information that may be nested within the code (specifically, substitutions of the form `%d` would need to be protected by `%%d`, etc). On the other hand, doing it in C would make some of the `call_data` information unavailable. It comes down to how much of the implementation side of this command needs to be made visible as a Hook object callback. The choice was made to do all processing by *tcl*.

Registering the Library

There is a single entry point into the extension library. This sets up the *tclMotif* environment, adds the Hook object callbacks, and loads scripts from the application resources. This is a single invasive call on an application, and source code may be needed to add this call.

This single entry point could be added to a widget such as VendorShell, and the Xt library rebuilt. If this is a shared library then these extension mechanisms will then be available to all applications using shared libaries without any need for recompilation.

Examples

Releases of Motif before 2.0 contained a demonstration program called `xmeditor`, which gave a basic text editor using the Text widget in a MainWindow with suitable File and Edit menus. In terms of functionality, it has been superceded by many free and commercial editors. However, its simplicity makes it a suitable candidate for extension. The intent in these examples is not to turn `xmeditor` into a fully-featured editor, but to show by easily understood examples how extensions can be made.

`xmeditor` from OSF is written in C as `xmeditor.c`. In the *tclMotif* distribution is a *tcl* version of this editor, `xmeditor.tcl`. The C version takes 1130 lines, the *tclMotif* version takes 230 lines (without comments).

The editor possesses no spell-checking capability, which would be useful when editing text. The editor was extended by adding in a new pulldown menu item with two entries, "Spell Document" and "Spell Selection". This runs the Unix `spell` program on the text, and shows misspelt words as a list in a SelectionBox dialog. Choosing a misspelt word highlights and selects the word in the text, so that typing any new text replaces the selection. A dialog button of Next moves to the occurrence of the misspelt word.

Three implementations were done of this extension: the addition of C code to `xmeditor.c`, the addition of *tcl* code to `xmeditor.tcl`, and the addition of *tcl* code to `xmeditor.c` using the extension techniques of this paper. Each created a new Cascade button, a new pulldown pane, two new buttons on this, and added callbacks that accessed the Text widget to get the text value. The misspelling dialog callbacks further manipulated the Text widget by setting highlight and selection.

The C extension took 440 lines, while each of the tcl extensions took 240 lines. The tcl code was syntactically simpler than the C. Both the C extension to C, and the tcl extension to tcl, required access to the source. The tcl extension to C required only the installation of the extension library to allow any extension to be made without further changes to the application.

The addition of menu items is a fairly "static" extension. It takes an existing widget tree and adds some new elements into this, with the new functionality more or less isolated into these new elements.

A more dynamic one was made also using spell checking, in an "immediate" mode. This makes more direct use of the existing widgets. During typing, or in more general cursor movement such as moving to end-of-line, there will be movement from out of words. This can be detected by attaching suitable code to the `XmNmotionVerifyCallback` on the Text widget already used by `xmeditor`. This is invoked on every movement of the primary cursor. By looking for whitespace between the old and new locations, a transition out of a word can be detected. Misspelt words can again be displayed in a dialog.

This was also implemented in C for `xmeditor.c`, and in *tcl* for both `xmeditor.tcl` and `xmeditor.c`. The C extension took nearly 500 lines, and the tcl extensions took 500.

Both of these extensions can be added as the event loop is run - the necessary elements already exist by then. It is however quite common for applications to delay creation of elements such as FileSelectionDialogs until they are actually needed. While this is not the case with `xmeditor` (all dialogs are created unmanaged before the event loop), the third example uses the `onCreate` mechanism to add callback code to the `okCallback` of the New/Open FileSelectionDialog

```
set fileDialog {.main.file selection dialog_popup.file selection dialog

onCreate {
   if {[string compare %widget $fileDialog] == 0 {
        $fileDialog okCallback {checkFileExtension %%value}
   }
}
```

The purpose of this is that when a new file is opened or created, the file extension is examined. Depending on the inferred file type, an editor-mode menu can be created, such as C-mode or tcl-mode.

These extensions use the facilities of *tclMotif* only. As part of the project, *replayXt* was folded back into *tclMotif*. This allows control of the GUI elements using sequences of commands which cause the widgets to execute actions using `XtCallActionProc()`. This inherits the uses from *replayXt* of test and replay, but also allows the possibility of complex "macro" sequences of actions.

For example, we could attach a macro to indent the current line when ctrl-I is typed:

```
.main.textSW.text modifyVerifyCallback {checkCtrlI %w %ptr %length}

proc checkCtrlI {w ptr length} {
    upvar 1 $ptr p

    scan $p %c j
    if {$j == 9} {
        $w callActionProc beginning-of-line()
        $w callActionProc insert-string("    ")
    }
}
```

While replayXt also contains a mechanism for recording simple macros, a suitable library-based mechanism for this needs to be defined.

Remote agents

The discussion so far has been of an application being extended directly by a ‘‘local’’ tclMotif script. In a distributed environment it is useful to see how a remote agent may act on on a client application. Examples of such use include remote control for impaired users, remote demonstrations, shared whiteboards, etc.

In the context of the current system, we would like to control applications by remote *tclMotif* programs run by agents. There are two ways that agents can run such programs: transmit the program to the client for execution, or execute the program directly and send suitable lower level commands to the client.

The next section discusses the first method using the ‘‘send’’ protocol and the following section discusses the second method using the x-agent protocol. This is followed by a preliminary discussion of security issues.

Send

There is a command in the Tk toolkit called `send`. This uses properties on the server to send a tcl command from an agent to a client running Tk. Tk clients handle `ClientNotify` messages specifically looking for such remote agent programs. When one is received the client executes it, returning results to the agent.

The "send" protocol handler has been ported to Xt as the tclXtSend library [Newmarch94c] so that a Tcl interpreter running in an Xt application can execute tcl commands sent to it. Such commands may be sent by any Tk application, or by the `xsend` applet included in the XScript system [Newmarch95a].

tclMotif links in the tclXtSend library, so that applications can be controlled remotely using this mechanism. No further work needs to be done.

The "send" protocol is a text-based protocol. This contrasts to other protocols such as X that are optimised to the byte-level. If high bandwidth was required this could be a problem, but is not here. This is because messages carry a high degree of content and are usually sent with time delays to slow the apparent execution to visible speeds.

RAP

The Remote Agent Protocol (RAP) is an initiative of the x-agent group, and aims to develop a protocol with a primary design focus as agents for visually impaired users [Edwards95]. The protocol is under development as a future X Consortium standard for remote agent control of applications. The emphasis is on *control* rather than *extension*, so it is unlikely to be complete for the purposes of this paper.

RAP extends the editres protocol to allow much greater information exchange both ways. It also uses the Hook object to give notification of widget creation/destruction and change of state.

The protocol defines a ‘‘DoActionRequest’’ to execute an action on a widget. This is sufficient for the general control of applications, such as emulating user actions. This can be used both for automatic demonstrations and replays.

For automated testing, some further elements are required: namely the ability to test widget state. In addition to GetValues, tests such as IsManaged are needed.

Whn it comes to extending behaviour, still more functionality is needed. This is because extensions make full use of the Xt (and to some extent, Xlib) calls.

For example, not only is state set by SetValues, but also by managing widgets, setting sensitivity, etc.

There are a number of convenience methods for widgets such as Motif's List that are not implemented by widget actions, but which can often be executed by a set of actions. It would be useful to build a library performing such a mapping.

The protocol is limited in the support for non-widget events, such as drawing into a DrawingArea. Rather than descending to a low level, it may be better to build widgets with support for drawing functions through actions. An alternative would be to use a set of widgets/gadgets that implement standard drawing items and are controllable by usual widget methods [Truong95].

The major lack from the extension point of view is the ability to create widgets within the application. Determining what classes of widget are available on the client side may be one problem. Even if such objects are created there remains the problem of associating application code with it - here the application would not be the client but the agent, and the code should execute on the agent side. A clean way of implementing this needs to be found.

Execution of programs on the agent side thus requires a great deal of addtional extension to a protocol such as RAP.

Security

Tk's "send" uses a server-based authorisation system (unless turned off at compile time). This assumes that applications with equal access to the server can control each other. This is quite reasonable, but conservative.

The x-agent protocol is built above the ICE (Inter-Client Exchange) protocol, and this can use a variety of methods to setup a connection that is assumed safe for the duration of the session.

The CCI (Common Client Interchange) protocol is an experimental protocol designed for control of Web applications, in particular, Mosaic. The control in CCI is limited to loading/recording URLs. Since Mosaic under X is a Motif application, there are advantages in bringing it under replayXt control, using an extension of the CCI protocol [Newmarch95b]. This allows, for example, demonstrations where a user of a browser can have another remote browser copy all actions such as selection of text, scroll up/down through a document as well as track URL accesses.

Security using CCI is virtually non-existent. Using replayXt to perform this control required a vetting of each individual action executed by a widget to ensure that the "Save As..." button could not be activated by a hostile remote agent. Such vetting was performed using a safe-tcl layer [Borenstein94]. A refined version of safe-tcl is being built into tcl version 7.5. The role of this will require evaluation for this system.

Conclusions and further work

This work has shown how the GUI elements can be extended and controlled without source code availability, probably in a different language to the original application. This can be done from scripts loaded by resource specifications, or by remote agents.

This has not been possible with Xt applications in R5 or before because the objects manipulated in the original application have not been accessible to the extension language. For example, a C application could not be extended using *tclMotif* because the application was unable to create *tclMotif* objects in a simple way.

This technique could be extended to other languages if they allow the necessary access to the event loop before (or after) executing each statement. If this is not available then the execution time of any program would have to be severely limited.

In an action-object system such as DTKSH this integration of application and *tcl* could have been performed using `XtNameToWidget()` to map between strings and widgets. In an object-action system such as *tclMotif* it requires the Hook object. In other toolkits a similar situation would probably be found.

This system is able to work because of the richness of mappings to widgets and their state, and the regularity of widget control mechanisms. It may be possible to extend this to other systems with such regularity, or to OO systems where the IDL can be queried.

For many applications extensions may require access to other internal data structures to be useful. This could only be done with this type of system if the application exported commands to the *tcl* interpreter for manipulation of the structures. This would require a far more proactive approach from the application, by defining *tcl* comands for use by extensions.

Availability

tclMotif is available from ftp://ftp.canberra.edu.au/pub/motif/tclMotif. It is also on X and tcl archive sites. Versions available after February 1996 contain the extensions discussed here.

References

[Borenstein94] N. S. Borenstein *Email with a Mind of its own: the Safe-Tcl Language for Enabled Mail* ULPAA 94, Barcelona

[Buehler92] K. Buehler *The xgen Application Generator: Constructing Motif GUIs for TTY Programs* pp 109-136, The X Resource, Issue 4, 1992.

[de Figueiredo95] L. H. de Figueiredo *The Making of BS, a Programmable Button Shell for X* X Advisor 1995, vol 1, 1993 (http://landru.unx.com/DD/advisor/)

[Edwards95] W. K. Edwards, et al *A Remote Access Protocol for the X Window System* pp 245-256, The X Resource, Issue 13, 1995

[Heller91] Dan Heller *Motif Programming Manual* O'Reilly and Assoc, 1991

[Java95] Java Home Page http://java.sun.com/

[Neumann93] G. Neumann and S. Nusser *Wafe - an X Toolkit Based Frontend for Application Programs in Various Programming Languages* Proc Winter USENIX Conference, 1993

[Newmarch94a] Jan Newmarch *tclMotif: a Binding of tcl to Motif*, XHibition94, San Jose 1994, ftp://pandonia.canberra.edu.au/pub/motif/tclMotif/

[Newmarch94b] Jan Newmarch *Using tcl to Replay Xt Applications* pp 137-144 Proc AUUG 94, September 1994, Melbourne, Australia, ftp://pandonia.canberra.edu.au/pub/motif/replayXt/

[Newmarch94c] Jan Newmarch *tclXtSend - using Tk's send for Xt* ftp://pandonia.canberra.edu.au/pub/motif/tclXtSend/

[Newmarch95a] Jan Newmarch *XScript: Shell Programming with X* pp 19-34, X Resource, issue 15, 1995, ftp://pandonia.canberra.edu.au/pub/motif/XScript/

[Newmarch95b] Jan Newmarch *Extending the Common Client Interface with User Interface Controls* pp 437-446, AusWeb95, Southern Cross University, 1995, http://www.csu.edu.au/ausweb95

[Ousterhout94] John Ousterhout *Tcl and the Tk Toolkit* Addison-Wesley, 1994

[Sall95] Ken Sall *The Motif FAQ* news:comp.windows.x.motif

[Stabile95] L. A. Stabile *Simplifying GUI Construction by Embedded Scripting* pp 43-70, X Resource, issue 15, 1995.

[Pendergrast94] J. S. Pendergrast, Jnr *The CDE Desktop Korn Shell* XHibition94, 1994

[Truong95] Tan Truong and Jan Newmarch *Structured Graphics using Motif Gadgets* unpublished, 1995

Author Information

Jan Newmarch is a Senior Lecturer in Computing at the University of Canberra. His Web page is http://pandonia.canberra.edu.au. Huimin Xu is working towards a Masters of Information Technology by research at the University of Canberra. Her email address is huimin@ise.canberra.edu.au.