# Using tcl to Replay Xt Applications

*Jan Newmarch*

Faculty of Information Science
and Engineering
University of Canberra
PO Box 1
Belconnen
ACT 2616

email: jan@ise.canberra.edu.au

Paper presented at AUUG94, Sept 1994,
Melbourne, Australia

*ABSTRACT*

Testing of X Window applications is often done using a low-level approach of synthesing X events, sending them to the application, and testing the result by comparing screen dumps. This paper describes a system that uses the object structure of Xt widgets to give a higher-level system. It uses the language tcl as the object action description language.

## 1. Introduction

In order to perform regression testing and to give automated demonstrations of X Window applications, it is neccessary to have some means of replaying X ‘‘input’’ events. That is, what would normally be user interaction with the application by mouse and keyboard must be simulated by some means. The X Window System supplies a simple means to do this in that actual X events may be prepared and sent to an application using *XSendEvent()* [1]. However, this is a rather low-level approach in some instances. It depends on windows being in particular locations and of particular sizes [2, 3]. Since such resources are often under the control of the user of an application through the X resource database, there is no guarantee that the events go to the correct place in a window.

Similarly, to examine the resulting system, comparison of screendumps may be used. This is even more fragile, as it depends additionally on fonts used, the colormap in use, etc. It is akin to comparing files of Ascii text which may differ if they have been shown on orange instead of green screens!

Applications based on the Xt Intrinsics use graphical objects called widgets, such as Lists, Labels, Text objects, etc. [4] The user interaction with such objects is at a higher level than that of clicking mouse buttons and pressing keys - the user selects a List item, or enters text into a Text object. The widgets themselves use this higher-level structure, by responding to actions such as ‘‘select item’’, ‘‘insert text’’, and so on.

In addition to this higher-level interaction, Xt objects are highly configurable, so that, say, a demonstration run by one user may have an apearance quite different to that run by another. Such differences are not only in colours and fonts, but may also involve size and geometric relation between component objects. Simulation methods that involve X events have to take this into account by suitably adjusting geometry of the events - it is not enough to just replay a set of events without modification.

This paper reports on a project that uses the object nature of Xt widgets to give a suitable replay language for events. The next section describes the components of Xt that allow this to be done. This language is based on the command language tcl [5], and this is briefly described in section three. Sections four and five describe how these are integrated to form the replay language. Sections six to eleven describe additional mechanisms to make this functional, and section twelve describes some implementation details.

## 2. Action procedures

Users get widgets to do things by use of the mouse and keyboard. The widget writer defines the effect of user actions by setting up *translation tables* that relate the user originated mouse and button events to widget actions, which in turn is related to widget code by *action tables.* The widget action functions will often invoke a set of callback functions. The application programmer will have created callback functions to reflect application functionality.

There is a set of action functions for each widget. In a well designed widget class everything that can be done to a widget is accessible through one of these action functions. For some widgets this set is small (eg for the Label widget described in Asente and Swick there are no actions) to very large (eg the Motif Text widget has nearly ninety covering actions such as scrolling, deleting, inserting, etc). The set basically forms the widget-specific set of object methods and gives it the characteristic behaviour of the class.

## 3. Tcl

Tcl is a language designed to be used as an embedded command language. It has simple syntax but a well-developed set of control structures. The syntax is a cross between C and the csh, with some syntactic problems of the shell cleaned up (such as [...] for grave ‘...‘ commands). A typical example reads a set of integers from standard input, one per line, and prints the maximum:

```
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"
```

An application is made ‘‘tcl aware’’ by creating a tcl interpreter and then calling this to execute a script of tcl commands. This script in turn may execute C code to read/write from files, process X events, or whatever else is desired. The language is extended by defining new tcl commands to the interpreter and registering the C code with it to execute when the tcl command is called.

The advantages of tcl over a proprietary language are numerous. Tcl is now becoming a commonplace language for embedded applications so the learning curve between applications is reduced. It is also an open language with published source code so that ambiguities or difficulties may be quickly resolved. The fact that tcl has full language capabilities allows for flexible use beyond initial planning. For example, in this system it can allow stress testing by placing replay statements within a loop, or arbitrary analysis of results on completion.

## 4. Widget naming

Widget names are translated into widgets using the *XtName()* function. This uses a widget as root and uses the Xt pattern matching for widget names. This consists of a dot ‘.’ separated list of widget names in the tree down to a widget, or a star ‘*’ pattern match of elements in this path. The root used in this system is the toplevel ApplicationShell widget that is used as a parameter to initialise the ReplayXt library.

## 5. Replay mechanism

The Intrinsics supply a function *XtCallActionProc()*. This function takes a widget and the name of an action and invokes the corresponding action function on the widget. In object terms, this calls a method for the object. This method may change the appearance and properties of the widget, and in addition will call application-specific callback functions. Thus this function allows full access to simulation of user activities. Simulating a set of user actions can be done simply by performing a set of *XtCallActionProc()* calls with suitable arguments.

Performing this set of action function calls in a simple manner is then the key to a simple simulation system. The set should be stored in an external file so that it can be easily modified for different simulations. Certainly, this would not be something that you would want to code in C! In addition, the mechanism must work simply enough that it can be used without alteration of the application between production use and such simulation.

It is easy to set up a mechanism using application resources to turn on and off a flag setting simulation. Tcl is used to satisfy the other requirements. An application creates a tcl interpreter and registers a new tcl command with it: *callActionProc*. This is linked to C code that takes two arguments, a widget name and an action, and performs the following:

1) finds the actual widget using *XtNameToWidget()*.
2) prepares suitable parameters for the *XtCallActionProc()* function.
3) calls *XtCallActionProc()*.

A typical sequence of actions would be

```
callActionProc rowcol.top_button Arm()
callActionProc rowcol.top_button Disarm()
```

which would call the two actions *Arm()* and *Disarm()* on the widget with name *rowcol.top_button*. The interpreter is initialised to read the commands from the external file and execute them.

As each action is executed the widgets will do something, including calling application code, which in turn may perform other actions, including creating dialogs or other widgets. During all of this the application and widgets must be able to respond to server events (such as Expose). Thus after an action is performed control must pass back to the Intrinsics event processing loop for proper processing of such events. The mechanism whereby this is done is described in the section on Implementation.

## 6. Preparation of X events

The *XtCallActionProc()* function has as one parameter an X event, which is normally the X event that triggered the action. In this system, of course, there is no such event, so an event may need to be manufactured. Most action procedures ignore the event, so for these nothing needs to be done. An example of where the event is used, though, is the *Activate()* action of the Motif PushButton. This action is usually called when the user releases the mouse button after having pressed it on the PushButton. If the mouse button is released inside the widget, the callbacks on the *XmNactivateCallback* list are called, otherwise they are not. The Motif implementation of this action examines the *x* and *y* coordinates of the event. In a similar manner, the Text widget determines what string was typed when an *insert-string()* action is invoked in order to insert the correct text.

Unfortunately, there is no documentation covering what exactly is required from an X event. Nevertheless, this system allows an X event to be generated with appropriate fields set. This is done by adding parameters to the *callActionProc* function which define the fields. For example,

```
callActionProc rowcol.top_button Activate() \
    -type ButtonRelease -x 0 -y 0
```

## 7. Warping the pointer

In order that the simulation have an adequate appearance, the mouse pointer should often be moved. For example, it should be moved over a

button before the button is pressed. The *warpPointer* command is defined in tcl which takes a widget and the *x* and *y* co-ordinates within the widget to move to.

## 8. Sleeping

For testing, it probably does not matter too much how fast the simulation runs. For demonstrations, it is rather critical that the actions be replayed at a suitably slow speed. A *sleep* command is available that pauses the replay of actions for the given interval. This is done in a way that allows normal processing of X events to occur, and is further described in the section on Implementation.

## 9. Getting resource values

Whn a user presses a mouse button over an object such as a PushButton, there is usually an attempt to get the pointer near the middle of the object, or at least away from its edges. It would certainly not look very nice to press a Motif PushButton at its very top corner (co-ordinates (0, 0)) as this would be well in the widget's border area. To move the pointer to the centre requires that the current dimensions of the object be known. The system contains a tcl interface to *XtGetValues()* to allow queries for resources such as width and height.

```
getValues $widget -width w -height h
```

stores in the tcl variables *w* and *h* the width and height of the widget. These can then be used in later calls such as to *warpPointer*

```
set mid_x [expr $w/2]
set mid_y [expr $h/2]
warpPointer $widget $mid_x $mid_y
```

In addition to preparation of actions and pointer movement, this facility is also needed for testing the state of widgets in the application to ensure that they have the correct values of the resources.

## 10. Gadgets

Gadgets were designed as lightweight objects to take memory and processing load away from the X server and place it back on the client side. Improvements in server technology have reduced the need for gadgets but they nevertheless remain in toolkits such as Motif. A gadget does not have windows, and it does not handle input events. It does not possess any actions since it does not handle input events, so the method described above does not work directly.

The means by which a gadget reacts to input events are not specified by the Intrinsics. Instead, in some manner, the gadget's parent container must handle them. This may be done in different ways depending on the toolkit and on the regularity of implementation within the toolkit. The remainder of this section discusses the Motif toolkit [6].

Motif defines a set of actions on the Manager widget which acts as a superior class for all Motif containers. These actions are specifically to handle gadgets. When a button press or other event occurs within the Manager it will examine its own translation table to call an appropriate action. For an action such as button press (using the default translation) this will call the action *ManagerGadgetArm()*. The purpose of this function is to determine if the event occurred within a gadget and if so to call an appropriate function within the gadget. Effectively, the Manager is acting as a translation table manager for the gadget.

In order to determine whether or not the event occurred within a gadget, the *x* and *y* fields of the event must be examined. In our system, we have no actual events since we are faking input, so suitable events must be created. This can be done exactly as above. There is one problem: these coordinates are with respect to the Manager widget, not internally to the gadget, so to direct actions to a particular gadget we must find its coordinates within the Manager. However, this can be done simply by getting the *x* and *y* coordinates of the gadget *g* within its manager *m*:

```
getValues $g -x x -y y
callActionProc $m ManagerGadgetArm() \
   -type ButtonPress \
   -x $x -y $y
```

## 11. Rule sets

tcl allows procedures to be defined to give structure to programs. This can be used to group actions into more meaningful sequences. For example, the following procedure can be used to perform a ‘‘button click’’ on a widget such as a Motif PushButton:

```
proc buttonClick {widget} {
   global DOWN_TIME

   getValues $widget \
      -width w -height h
   set mid_x [expr {$w / 2}]
   set mid_y [expr {$h / 2}]

   # move pointer to middle
   warpPointer $widget $mid_x $mid_y

   # press
   callActionProc $widget Arm() \
      -type ButtonPress \
      -button Button1 \
      -x $mid_x -y $mid_y
   sleep $DOWN_TIME

   # release
   callActionProc $widget Activate() \
      -type ButtonRelease \
      -x $mid_x -y $mid_y
}
```

Procedures such as these could be stored in a rules file which would be loaded into the tcl interpreter before the data file is read.

## 12. Non-structured objects

Xt uses widget and gadget objects that can be handled as described above. Unfortunately, some widget sets use widgets that have internal structure that is not visible via Xt objects. A tyical example is the Motif List widget. What appear to the user to be component objects, the elements of the list, do not appear as objects within this widget.

When the user selects an item, the List widget has to perform internal calculations to determine which item is actually pressed. This calculation is based on such things as the number of items showing and the total visible height of the List. In order to prepare a suitable action for such an object, we need to reverse this calculation in order to determine the correct y value to give an item. This is regrettable for many reasons.

The previously described *getValues* method is also needed here, to extract resource values from the List. It is best to capture this in a procedure as in:

```
proc indexToY {w n} {
   # calculate the Y co-ord in widget
   # w for an index n in the list
   getValues $w \
      -height h -itemCount count

   set y [expr {(2*$n - 1) * $h / (2*$count)}]
   return $y
}
```

## 13. Implementation

The naive approach is to simply read and execute a file of the actions that the application has to perform. This is quite unsatisfactory. The tcl *source* command is an atomic command as far as processing X events from the server, so that while the tcl commands in the source file are being executed, no X events are processed. At best they would all be queued. Then after execution of the actions was over, all the queued events would be processed. The application would initially appear to be frozen, and then respond in a flurry of activity.

To avoid this, it is necessary to allow the X event loop to be called frequently to process queued events. The most common ways of doing this

both involve breaking a large atomic process into a set of smaller processes. Each of these can be called using timer events, with each process scheduling the next, or by using work-procedures, where the flow from one to the next is similarly maintained.

In order that the application does not just ‘‘rush’’ through its actions, it is neccessary to delay actions and run them at timed intervals. This cannot be done using the system call to sleep as this would stop the application from responding to X events while waiting. This precludes workprocs, and makes timers look attractive. Unfortunately there is no simple way to break evaluation of a set of commands by a tcl interpreter into a set of separate evaluations.

An initial attempt was made to read the file, stopping reading and evaluating when a complete command is ready (i.e. read a command, execute it, read another, etc). This also proved unsatisfactory. A complete command may be a procedure call containing many other commands, and this approach is unable to penetrate to the internal structure. This means that a set of commands containing delays may work ok, but placing them in a procedure would lose control of the delays. Similar behaviour occurs with any other compound structure such as loops.

The alternative to these is to take control of the X event loop explicitly. That is, at suitable points, call the Xt event processing commands. Tcl contains a ‘‘trace’’ mechanism which calls a tracing function before executing any command. By making this function process any queued X events, the finest granularity of both tcl and Xt is gained. Execution delays can be managed without degrading response by entering an event loop for the period of the delay, with the delay end signalled by an Xt timer event.

## 14. Send

The most common use of this system is expected to be by reading a file of rules and a file of replay commands. This is a rather static situation.

The Tk widget system introduced a communications primitive called *send* between Tk applications. This allows one Tk application to send tcl commands to another for execution in the second application. An example use is a debugger sending commands to an editor to step through the file.

The *send* command has been ported to Xt by the author as a separate library. This is linked in with ReplayXt so that a Tk application or an Xt application with *send* can send commands to an application linked with this system. This will allow one application to invoke commands, which here would most likely be actions, and also to query the application for resource values. This allows an easy way for, say, a debugger written in Tk to control an editor written in Motif.

## 15. Status

The software has been released into the public arena, and is available from sites such as ftp.x.org. The version at the time of writing is the initial version, 1.0.

## 16. Future work

The Intrinsics supply a function *XtAppAddActionHook()* which can be used to attach a ‘‘record’’ function each time an action is invoked. This allows an application to record for later play. Experiments are being performed to determine the best way of outputting a script for later replay. Dumping actions using events with hard-coded *x* and *y* values, for example, is quite easy, but suffers the problems of resizing. It is necessary to extract context information so that the event description will really be ‘‘half way down’’ the widget instead of 20 pixels, say.

In a similar way, comparison of pixmap dumps of windows is quite feasible but too restrictive. A means is needed to perform ‘‘feature extraction’’ for meaningful comparison.

Some applications use multiple toplevel Application shells. These are typically applications running across multiple displays. A mechanism to allow switching between widget trees is under investigation.

Finally, the capability to define rules by procedures leads to a search for suitable rules. Some,

such as the *indexToY* for Motif List are quite straightforward. Others may be less so.

## 17. Conclusion

This paper has described an object-oriented means of controlling the replay of Xt-based applications using a standard and readily accessible command language. This allows for demonstration or for testing of applications in a simple manner.

## 18. References

[1] R. W. Scheifler, J. Gettys and R. Newman, *X Window System - C Library and Protocol Reference,* Digital Press, 1988

[2] A. Azulay, *Automated Testing for X Applications,* X Journal, May-June 1993

[3] L. R. Kepple *Testing GUI Applications,* X Journal, July-August 1993

[4] P. Asente and R. Swick, *X Window System Toolkit, the Complete Programmers Guide and Specification,* Digital Press, 1990

[5] J. K. Ousterhout, *Tcl: an Embeddable Command Language,* Proc USENIX Winter Conf, 1990

[6] Open Software Foundation, *OSF/Motif Programmer's Reference,* Prentice-Hall, 1993