TkReplay: Record and Replay in Tk

Charles Crowley

Computer Science Department
University of New Mexico
crowley@cs.unm.edu

Abstract

Record and replay of user interactions with a GUI are useful for regression testing and demonstrations. Recording is implemented by intercepting each user action at some point in its processing and saving it in a script file. During replay, the script is read and the user actions (or their effects) are emulated. In X windows, one program can intercept user input events before they get to another program by putting a "wrapper" around the program. This level of recording is sensitive to changes in window position, fonts, etc. Recording events at the widget level is more robust and closer to the semantics of the program than recording at the X event level. The TkReplay program does this for the Tk toolkit. A number of factors have to be considered when implementing record and replay and these are discussed in the paper. Recording is implemented in Tk by modifying each binding to call a recording program. TkReplay operates as a separate program and uses send to communicate with the Tk program being recorded. Hooks have recently been added to the X toolkit to allow implementation of record and replay at the Xt widget level.

Introduction

Normally a program with a GUI is operated interactively by a user who sits at a workstation and enters input using the mouse and keyboard. In this paper we will consider a facility that allows you to record a sequence of user inputs in a script file and to replay them at a later time. During replay, it is not necessary to have a user at the workstation entering input, instead the input that was recorded will be replayed exactly as it was originally entered. The program will receive this input and repeat the responses it made to the original input. This is akin to running a character-based program in "batch" mode where all input comes from a file.

There are (at least) three reasons why one would want to record and replay a user interaction: for demonstrations, for regression testing and for scripting.

We can record a *demonstration* of a program in a script and run the program from the script to show the capabilities of the program. The demonstration might be used in a help system, in a tutorial or in a marketing presentation.

Regression testing involves running a set of tests on a program each time any change is made to make sure that the changes do not affect other functions that were already working. Regression tests for a program should be recorded so that it is easy to repeat the tests after each change to the program. Automated testing of GUIs is probably the most important reason for record and replay.

A *script* automates the performance of a task with a program so that you do not have enter all the commands interactively each time you want to perform the task. But many programs can only be run interactively and do not have a scripting language. A record and replay facility can act as a scripting language for such a program.

In this paper I will look at the issues in implementing record and replay in GUI systems and then look at the implementation of TkReplay, a tool for doing this in Tcl/Tk programs. I will look at the problems that were encountered and how they were handled, the things in Tk that made it hard to implement record and replay and briefly discuss implementing record and replay in the X toolkit.

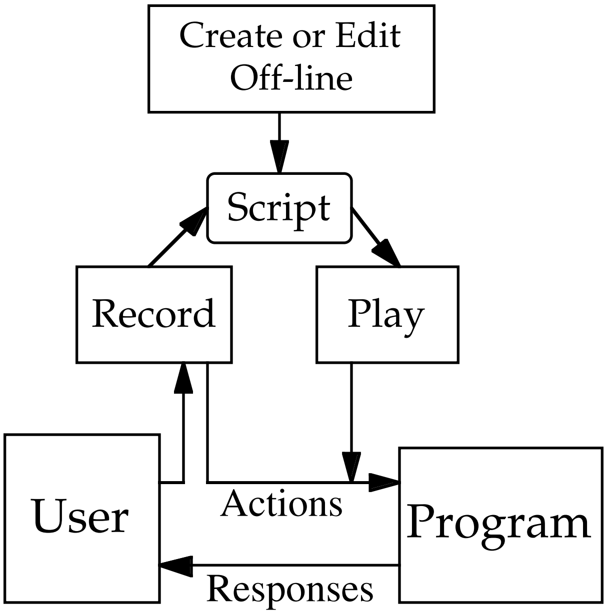
Some definitions

The program doing the recording (or replaying) will be called the *replay application*. The program being recorded (or replayed) will be called the *target application*. User input takes the form of a sequence of *actions*. A record of these actions is called a *script* which is stored in a file.

There will be another use of the word "script" in this paper. A Tk binding attaches a script of Tcl commands to an event in a widget. We will always refer to this as a "Tcl script" to avoid confusion with a script of actions.

Issues in Record and Replay

To implement record and replay, we must intercept and record user actions and then, when replaying, regenerate these actions and send them to the program as if the user had taken the actions. It is necessary to record enough information about the user's action so that the replay mechanism can regenerate the action and so that it will produce exactly the same response in the program. The following diagram shows the parts of a record and replay system.



The recording mechanism intercepts and records the events and then sends them on to the program so that the user can see the results of these actions. Since the script is a file, we could create the script directly instead of recording an actual interaction or we could edit a script so that the replayed interaction is similar to, but not exactly the same as, the originally recorded actions.

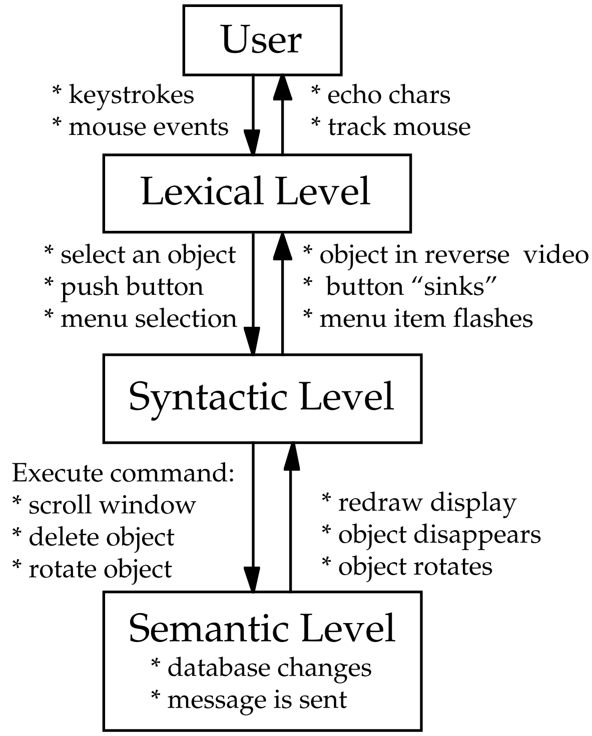
The issues to be considered are:

- What to record
- What to replay.
- How much to modify the target application.

Levels of Feedback

In order to answer the question of what to record and replay we will discuss the types of feedback present in GUIs. It is not necessary to replay all of this feedback. It is useful to divide the processing of user input into the three levels taken from linguistic analysis: lexical, syntactic and semantic.

The following diagram shows these levels. The downward arrows are inputs to each level and the upward arrows are the feedback from each level. Some examples of each are given in the diagram.



The *lexical level* consists of mouse button press and release events, mouse motion events and keyboard key press and release events. The input processing at this level provides lexical feedback consisting of moving the mouse pointer on the screen (feedback for mouse motion) and echoing characters (feedback for keystrokes). There is generally no lexical level feedback for mouse button presses or releases (but they usually produce some syntactic level feedback from the syntactic level inputs they generate). Lexical level input events are then processed at the syntactic level to generate syntactic level input events.

The *syntactic level* is tied to the widgets displayed on the screen. Syntactic level inputs are things like: selecting an object, pushing a button, dragging a scrollbar slider or selecting from a menu. The syntactic feedback from these events would be: the object is shown in reverse video, the button changes relief (it looks as if it is pressed in), the scrollbar slider follows the mouse pointer and the menu items change color to follow the mouse pointer. Some syntactic level events generate calls to semantic level actions.

The lexical and syntactic levels involve the specification of the action to perform. At the *semantic level,* the action is finally performed via calls to the application. These calls are called *semantic actions* and generally they modify the data the program is managing. Semantic actions are commands like delete a shape or rotate a shape. The semantic feedback from these actions involves an update of the screen representation of the data, that is, the updated display with the object deleted or rotated. In addition, there are other semantic actions (updates to the program database, messages sent to other processes, etc.) that are not immediately represented on the display

One or more lexical level input events generate a syntactic level input and one or more syntactic level inputs generate a semantic action. The lexical and syntactic events, and their feedback, occur before the semantic action, which may generate its own semantic feedback.

The primary purpose of user actions is to cause semantic actions to be executed. The feedback allows the user to see what has been done, however a replay mechanism may not need to replicate all levels of feedback. It must, of course, replay the semantic actions and the semantic feedback is produced by the program as a side effect of the semantic actions. The syntactic feedback is not necessary for scripting or for testing (unless we are testing the syntactic feedback itself) but it is important in a demonstration so the people seeing the demonstration can follow what is going on. They will see the buttons get pressed and then commands get executed. Lexical level feedback is not strictly necessary, but it is nice for demonstrations since it adds to the realism of the replay.

Changing the Target Application

Ideally, the replay mechanism should not make any changes in the target application and a single replay application should work with all programs. Unfortunately this is not always possible and so our replay mechanism will execute some code in the target application. However, it will not be necessary to scan and modify the entire source code of the target application in order to do record and replay. We will provide a generic mechanism that we can add to a target application that does not require changes in the code for each widget or each event binding. We will require the target application to make a single call to the replay application to set things up.

Infallible Record and Replay is Impossible

I should point out that perfect record and replay is impossible. Up to now, I have been making an implicit assumption that the behavior of the program is determined by the user input. If that is the case, then record and replay is possible, but, this is not always the case. For example, suppose that a program displays a message every hour on the hour. This occurs as a result of the system clock ticking and is not related to any specific user input. The hourly display might happen during recording but it will not happen during replay unless the recording and the replay are executed at exactly the same times.

A perverse (or very clever) program might look at the clock and create one kind of interface in the morning and another kind of interface in the afternoon. We cannot reproduce this in a record and replay system unless we can record and reproduce the system time. Normally a program gets the time from the system. We could probably intercept these calls and fool the program but it is not possible to control everything about the environment of a program.

Let’s take a more common example. Suppose you record a script that calls up a file selection box and selects a file. During replay the file system might be different and the file selected during recording might have been deleted. Replaying this script is no longer possible.

The lesson here is that you have to consider all the inputs the program uses (not just user inputs) and control all of these which might affect the operation of the program.

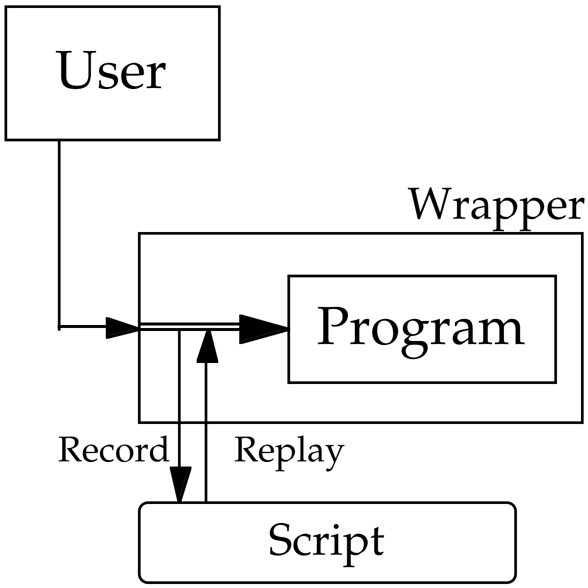
Record and Replay at the Lexical Level

One can implement record and replay by recording events at any one of the lexical, syntactic or semantic levels. Each of these options has advantages and disadvantages. In this section we will look at record and replay at the lexical level. In a later section we will look at record and replay at the syntactic level. Finally, when we discuss record and replay in the X toolkit, we will look at record and replay at the semantic level.

Character Based Programs

Some programs are character based (that is, their input is just a stream of characters typed by the user). The user types and the program responds. A script for a character-based program is a file containing the characters typed. Replaying a script is done by running the program with the input taken from the script file instead of the keyboard. In UNIX, and most modern operating systems, this is trivial to do with the input redirection facility [11].

Recording is a little harder. The general solution is to put a "wrapper" around the program. A wrapper reads the input, records it and passes it on to the real program you are running. This is also easy to do in most operating systems and it can be done generically so that a single wrapper program will record keystrokes for any program. The following diagram shows the wrapper strategy.



The wrapper approach is a little harder for full-screen character mode programs. Some programs will only run when attached to a "real" terminal. Pseudo-ttys were developed to deal with this problem [12,7].

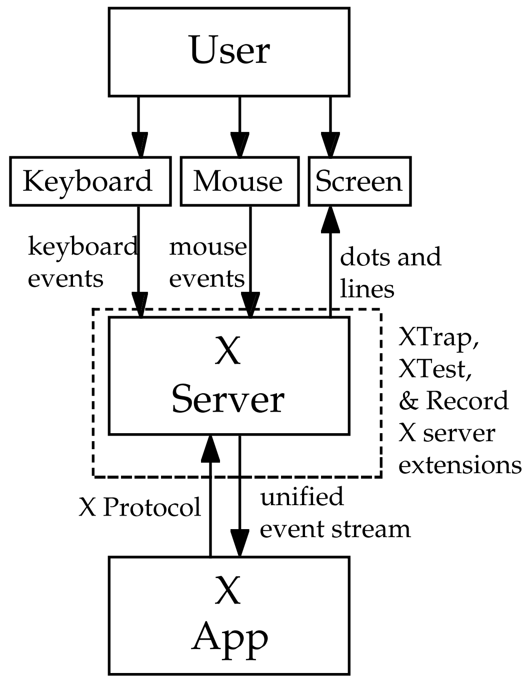
GUI Programs

In GUIs, the lexical level (for input) consists of the keyboard and mouse events. This level deals with screen pixels and low level user input actions. The syntactic level is the widget level and events are grouped by the widget to which they are directed. We can record and replay at either of these two levels. The wrapper model is restricted to the lexical level.

The Wrapper Model in X

The wrapper model can be used in the X window system. You must capture the stream of events going to the application (there are hooks to allow you to intercept events), record the events in a file and later replay them. There is an X library procedure that allows you to generate and send an event, so you can simulate mouse and keyboard input.

The basic X server allows you to do basic record and replay of X events but there are several technical problems. As a consequence of these implementation problems there have been several extensions to the X server that have been proposed and implemented over the years that allow an easier and more complete implementation. The XTest extension [4] is sufficient to implement a replay mechanism and the record extension (still a proposed standard) [13] allows for recording. These extensions are basically wrappers around the X server that allow the interception and injection of events. The diagram below shows the structure of events in the X window system.

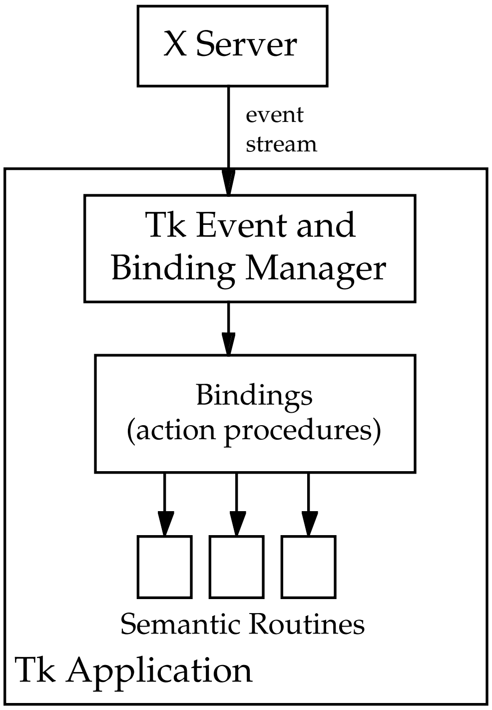


The problem with the wrapper approach is that it is fragile [2,6]. The mouse events are based on absolute screen coordinates. If the windows move, then the replay may not work. It is also sensitive to other changes. For example, the size of the borders placed around the window by the window manager can make a difference. Slight changes in the layout of the program may cause the replay script to fail. Also, many user interfaces are user customizable. The user can decide which fonts to use for example. A different size font might move things around.

The wrapper approach works at the lexical level since that is the level of X events. The wrapper records a mouse click at a certain screen location, not over a certain button. If the button moves, the location of the event will not, because there is nothing to connect the event and the button. Nevertheless, this approach does work if you are careful. There are testing tools based on this approach.

Implementing Record and Replay at the Tk Widget Level

The general X wrapper approach will work with any X program including Tk programs. Alternatively we could record and replay at the syntactic, or widget, level. The widget level is implemented inside the application with libraries, so we must make changes to the program itself in order to implement record and replay at this level. The following figure shows the event handling model for Tk. The “semantic routines” are Tcl scripts.



Tk directs events to widgets. The basic mechanism for handling events in Tk is the binding mechanism [9]. A binding specifies an event sequence and a Tcl script to run when the event sequence occurs in the widget. Some Tk widgets also have callbacks (called command options). We will discuss them later.

In the next few sections I will examine various issues in the implementation of record and replay in Tk.

Rebinding Widget Bindings

Recording at the widget level in Tk requires that we intercept each call generated by the binding mechanism. There is no way to do this centrally, so we have to do it by changing each binding to ensure that the recording mechanism gets called. The following Tcl/Tk code will rebind all the widgets for recording.

```
proc RebindAllWidgets {} {
  RebindWidgetAndChildren .
}
proc RebindWidgetAndChildren {w} {
  RebindEvents $w
  foreach child [winfo children $w] {
    RebindWidgetAndChildren $child
  }
}
proc RebindEvents {w} {
  global Bindings
  # find all the events that have an
  # associated binding
  foreach tag [bindtags $w] {
    foreach event [bind $tag] {
      # get the binding for this
      # tag and event
      set binding [bind $tag $event]
      # remember the binding for
      # later use
      set Bindings($tag,$event) \
        $binding
      # find out which % fields are
      # used in "binding"
      set percentFields \
        [FindPercentFields $binding]
      # rebind to the event to our
      # event handler which will
      # record the event, do the %
      # substitutions and call the
      # original script
      bind $tag $event "RecordEvent \
        $tag $event $percentFields"
    }
  }
}
```

We start at the root and visit all the widgets in the interface. For each widget, we find all the tags bound to it. Then we find each event that is bound to the tag and rebind it. For each binding, we save the original script in a table and rebind the event to call our recording procedure, which records the event and calls the original script.

This process catches all the class bindings since they are found in the `bindtags` list for widgets of that class. We remember what we have already rebound and only rebind each tag once. Widget bindings are just a tag with the same name as the widget.

Capturing X Event Fields

The Tk event handling mechanism uses %-fields to transfer X event data into the binding. It looks for strings of the form “%x” (where “x” is some letter) and replaces them with the appropriate field from the X event structure. We must duplicate this process when we call the event handler. In order to do this, we must capture any %-fields required in the binding. We do this by scanning the binding and recording which %-fields are present. `RecordEvent` will substitute the %-fields in the binding before it is called.

`FindPercentFields` returns a list of the form `{{W $W} {x %x} {y %y}...}` where each required %-field is represented. The binding mechanism will fill in the values and the record or replay code will take care of inserting the %-fields into the Tcl script of the binding.

Capturing the "current" object on a canvas

Canvases must be treated specially because canvas bindings often make use of the "`current`" object on the canvas, that is, the object that is directly under the mouse pointer. During recording we capture the object id of the `current` object. It is not possible to set the `current` object except by actually moving the mouse pointer over the object. What we do during replay is replace all instances of the string `current` in the binding with the id of the `current` object captured during recording.

Handling bindtags

In Tk4 each widget can have a set of associated tags and each tag can have events bound to it. So a single event might activate several bindings and several Tcl scripts may be executed. It is possible to break the chain of Tcl scripts by returning a break return code from a Tcl script called as an event handler. During recording we are executing the Tcl scripts ourselves so we have to capture and pass on the return code so the program will work as expected. During replay we execute the same Tcl scripts as we executed during replay and ignore the return codes. We do not give a Tcl script a chance to change its mind and return a different error code on replay than it returned during recording.

Callbacks

Several Tk widgets have command options or "callbacks" as they are called in X. The Tk widgets that have command options are: button, checkbutton, menubutton, radiobutton, scrollbar, scale, menu (`postcommand`) and menu entries. All of these callbacks (except for `postcommand`) are implemented with bindings so redefining the bindings automatically handles the callbacks.

The one exception is the `postcommand` option that is called just before a menu is posted. This gives the application a chance to modify the menu according to current conditions. This callback is called as part of the `post` subcommand to the menu widget. Since the `post` command is almost always executed in a callback we do not have to worry about redefining it.

Internal Widget Bindings

The canvas and text widgets each allow bindings to internal objects and have their own separate binding systems. These internal binding systems are virtually identical to the main binding system and are handled in a similar way. When we rebind a canvas or text widget, we first rebind the external bindings and then we go through all the internal bindings and rebind those also.

One detail that makes this harder is that canvas widgets allow bindings to both canvas objects and canvas tags. But there is no way to enumerate all the tags in a canvas. (This is done with "`$text tag names`" in the text widget.) The workaround is to enumerate all the objects and accumulate all the tags associated with these objects.

Handling Dynamic Changes

Any time Tcl code is executed it could change the bindings of any widget, create widgets or destroy widgets. Our recording mechanism is based on having rebound all bindings of all widgets. If a binding changes or a widget is created then we have to redefine the bindings before it gets called, so we have to detect these changes and make the necessary rebindings. The simplest way to deal with this is the redefine the `bind` command and all the widget creation commands.

The new widget creation commands will call the original widget creation command and then call a procedure to rebind all the tags of the new widget. There is one detail that we must consider. The `frame` and `toplevel` commands are implemented with the same code which looks at the first letter of the command name. If this first letter is "t" then a toplevel is created, otherwise a frame is created. So we have to be sure to rename `toplevel` to another name which starts with "t'.

In order to catch any internal rebinding in canvas and text widgets we have to rename the individual widget command (whose name is the path name of the widget) also. When this command is called we see if it is a bind subcommand and, if it is, redefine the binding.

Mouse Pointer Warping

If you are replaying a demonstration, you may want the mouse pointer to move just as it did when you did the recording. This involves two things, knowing where to move the mouse pointer and actually moving it. There is an X library procedure called `XWarpPointer` that moves the mouse pointer, but pointer warping is a controversial subject. Many human-computer interface experts advise that a program should never warp the pointer because the pointer should be under the control of the user. In fact, the documentation for the `XWarpPointer` command cautions that this command can confuse the user and should only be used in exceptional circumstances. As a consequence (or maybe for other reasons), Tk does not have a command to warp the mouse pointer. I had to add that functionality as a new Tk command. The problem with this is that the replay system will not work with the regular wish but needs a custom wish with pointer warping added.

One reason you want to move the mouse pointer is to call attention to what is happening during the replay. There is a version of `TkReplay` that does not use pointer warping but instead has a small window with a red arrow in it that moves around and points to the widget where the next event takes place.

It is easy to know where to move the pointer because the X records the x and y coordinates of the mouse for all mouse events. We can capture these with %-fields and use them to know where to warp the mouse pointer.

I should note that it *is* irritating to have the pointer moved around for you. During the debugging of the program I could not stop replays because I could not get control of the pointer long enough to set the focus and kill the application. I had to put in a special binding to stop the replay when a mouse button is clicked.

Name Conflicts

`TkReplay` defines several new procedures and a global array in the target application and has to worry about name conflicts. It uses names that end in "__rd". This string must be changed in the file to be sourced if there is a conflict with other names in the target application. It could try several names and see which ones are not being used but even this will not work in all cases. A program might define new names at any time and these might conflict with any name you choose, even if there was no conflict when you first defined the name.

Potential Deadlocks

Window systems are event oriented and work by calling application code when user events are detected. Sometimes it is necessary for user code to wait for a specific event to occur, usually the answer to a question in a dialog box. Tk provides this facility with the `tkwait` command that allows a you to wait for an event inside a Tcl script. This is implemented by running a local event loop.

The `tkwait` facility can interact badly with a replay facility. if you are not careful it is easy to get into deadlocks.

The replay application replays an action by sending it to the target application using the Tk `send` command. The `send` command sends the command given in its arguments and blocks while it waits for a reply. If the command it sends executes a `tkwait` then it will not return and the `send` command will not complete. Because of this problem we cannot execute the binding directly but instead use the `after` command so that the `send` can complete. But then the completion of the `send` command no longer indicates that the binding's Tcl script has completed so we have to send a response back to the replay program when the Tcl script has completed. But we have observed that the binding may not complete because it calls a `tkwait`. To solve this problem we have to set a timeout that will send a reply back after some time delay if the script has not completed. It is easiest to have both the timeout and the command send replies. The replay program accepts the first reply as signal to move on the next event to replay and ignores the second reply.

Here is the sequence of events when an action is replayed.

1. Get the next user action to replay.
2. `Send` the action to the target application.
3. The target application schedules the action using the `after` command, starts a timer (also using the `after` command) and completes the `send`.
4. When the action is complete a completion message is sent to the replay application.
5. When the timeout occurs another completion message is sent to the replay application.
6. The replay application continues after it gets the first completion message. It will ignore the second completion message that it will get later.

Let's look at the code to handle this. When an action is being replayed the replay program sends a command that calls the `ReplayAction` procedure:

```
proc ReplayAction {uid evid subs} {
  after $timeout send $replayApp \
    [list ActionEnd $uid]
  after 1 DoAction $uid \
    $Bindings($evid) $subs
}
```

The `uid` is a unique identifier assigned to the action dynamically by the replay application. It is used to identify the action in the "action completed or time out" messages that will be returned. The `evid` is the subscript in the `Binding` table where the code for the binding was saved. The `subs` are the %-field substitutions to make.

First we set up the timeout and then we schedule the action itself. Then the procedure returns and releases the `send`. The replay application then waits for the action to end or the timeout (whichever comes first). The `DoAction` procedure looks like this:

```
proc DoAction {uid action subs} {
  RealDoAction $action $subs
  send $replayApp \
    [list ActionEnd $uid]
}
```

So two `ActionEnd`s are sent for every action and three `send`s are required for each action. The replay application looks at the `uid` in the `ActionEnd`, ignores old replies and waits for an `ActionEnd` for the current action.

The TkReplay Program

I have implemented record and replay for Tk4.0 in a program called TkReplay (there is also an older Tk3.6 version). It takes into account all of the considerations described in the previous section. It records user actions in a script which can be saved in a file. The script can be later read in and replayed. You can load an existing script file, record additional actions (which are added to the script), delete actions and save the modified script. Scripts are in a simple ASCII format and can be edited and combined with a text editor

The first step in recording a script is to start the target application and "connect" to it. When TkReplay connects to an application it sends it a command to `source` a file of Tcl procedures and commands that redefine all the bindings and rename the commands TkReplay must monitor. Both loading and connecting can be made part of the script so that replaying the script will automatically load and connect to the target application.

It is possible to connect to several applications at the same time and record a combined script that includes user actions from two or more target applications.

Once you are connected to an application you can start recording. Actions show up in a list box as they are recorded. After you stop recording you can rewind the script and play it. You can start from any place in the script by selecting that action in the list box.

Adding Comments and Pauses

If you are recording a demonstration then you may want to display comments on what is happening in the demonstration. You might want comments for regression testing as well to inform the tested what to look for during replay.

TkReplay has a facility to display a comment after any event in the demonstration and remove it after some later event. The comment is in a popup text window.

There is also a command to add a pause in the replay.

Megawidgets and New Widgets

Megawidgets that are built up from existing Tk widgets work automatically with TkReplay but the scripts will not reflect the megawidget structure. The Tix widgets, for example, work with TkReplay.

TkReplay depends on redefining each binding and so it must know when widgets are created and when bindings are redefined. So new widgets must be added to TkReplay by hand. For widgets with no internal bindings, this consists of adding their name to a list. For widgets with internal bindings, custom code must be added to handle the internal bindings. The pad widget [3] is an example of this.

Problems with TkReplay

There are a few problems with the timing of the replays. TkReplay records the time delay between events and delays for that period when replaying the actions. But the replay mechanism itself takes time which cannot be easily predicted and so the replay is always slower than the recording. The replay will be affected if there are parts of the program that depend on timings. For example, suppose you have a button that autorepeats as you hold it down. The down time is the elapsed time between the button down event and the button up event. This elapsed time will be longer in the replay and the button will autorepeat more times in the replay. This could affect scrolling.

Since we only record mouse positions when events (like enter, exit, button down, button up) occur the mouse motion seems a little jerky.

Record and Replay with Xt Widgets

Most X widget toolkits, like Motif, are built on an intermediate layer called the X toolkit (or Xt) layer. This is a layer of routines to support the implementation of widgets. The Xt layer has a mechanism for binding user actions to application code that is based on *action procedures* and *translation tables*. A translation table is a text description that binds input events to action procedures [1]. It is very similar to the Tk *bind* mechanism. An action procedure is a C procedure that the application or widget has packaged so that it can be called from a translation table

Given these close analogies between Xt/Motif and Tk, we could use the same strategy we used in Tk to implement record and replay. This requires the same introspection facilities and hooks available in Tk. That is, we have to be able to traverse the widget tree and change all the bindings and we have to detect when bindings are added or changed and when new widgets are created. These facilities have recently been added to Xt in an extension to X11R6 called the Remote Access Protocol (RAP).

The Xt level also offers a facility to allow and easy implementation of record and replay at the semantic level. There is a specific hook that is exactly what you need, a procedure called `XtAppAddActionHook`. You pass it the name of a procedure that will be called just before any action procedure is called. Thus we do not need to rebind individual bindings, a single call does it all. Of course, recording at the semantic level of action procedures will not allow you to record or replay and lexical and syntactic feedback that occurs before the action procedures is called.

Another problem in Xt is the existence of event handlers. Event handlers are the X level mechanism for responding to user events. All other mechanisms (such as translation tables and Tk bindings) are implemented using event handlers. Programmers are discouraged from using event handlers directly and most Xt level applications do not use them. But if a program does use event handlers, any hope of recording all events is lost because there is no hook to get control before event handlers and there is no way to find out what event handlers are in effect or to override them.

Other Uses of Recording Hooks

The same hooks needed for record and replay can be used to create an alternative interface to a program, for example, an alternative interface to a GUI program for blind users [5].

The user interface of a program is an external model that is a reflection of an internal model that the program implements. The internal model is the important one and the user interface is a way of presenting that internal model to the user. The user interface is a way to inspect the internal model and to perform operations on it [10].

The widgets we normally use are appropriate for an ideal user with good vision and the ability to use a keyboard and a mouse easily. If a user does not fit this profile then the normal widgets might not be appropriate. What is important is to provide an interface to the user that reflects the internal model of the program. Suppose we had a blind user. It would be possible to redesign the user interface of a program to use sound and touch and to effectively present the internal model of the program.

But there are a range of possible disabilities and it is not feasible to change all programs to best suit a wide range of users. The best compromise is to provide generic methods of translating the normal Tk interface to one appropriate for a particular class of users, such as blind users. The generic mechanism can transform the interface into one based on, for example, sound and touch. There are many ways to do this and experimentation about the best way to do it is appropriate.

Implications for Tk

It was fairly easy to implement record and replay in Tk because two important features of Tcl/Tk. First, Tk is introspective and allows us to ask just about anything about the current state of the interface. Second, Tcl is dynamic and allows us to redefine the procedures we want to monitor and add the hooks we need. We can rebind existing widgets easily because Tk will tell you the current bindings. This means that we do not have to get control when an application first starts but can connect to it at any time. Because we can redefine key procedures we can find out when important events occur (that is, widget creations and new bindings) and deal with them.

It would be useful if Tk added a function that is equivalent to the Xt function `XtAppAddActionHook`. This would allow a very simple implementation of record and replay at the semantic level. It would allow you to define a function that is called just before a binding script is about to be called. It should pass you the necessary information, like what the event is, what widget it was in, what the binding is, and have a way to get the X event fields. The return value of the procedure would determine whether the binding was called.

It would be a good idea to implement frame and toplevel with two different C procedures so that people would be free to rename `toplevel` to whatever name they want.

A few additional features would be handy. These include: the ability to enumerate tags in a canvas, the ability to set the current object in a canvas, and the ability to warp the mouse pointer.

Other Work

There has been work on record and replay at the X level for several years and this has resulted in the X server extensions to support record and replay. There are several tools for testing GUIs that use these extensions (e.g., XRunner).

Record and replay at the widget level are just starting to get attention. Jan Newmark [8] has implemented a replay mechanism for Tk and has recently extended it to do recording (using *XtAppAddActionHook*).

Summary

In this paper I have examined at the general issue of record and replay in GUIs, looked at the various levels at which record and replay can be done, and looked at the issues that must be addressed when implementing this facility in Tk. In the course of the discussion I have identified a few changes to Tk that would make implementing record and replay simpler.

Availability

TkReplay is available from `ftp://ftp.cs.unm.edu` and from the Tcl/Tk archive.

Acknowledgments

Some of this work was supported by a contract from Sandia National Laboratories. I want to thank Gene Hertel of Sandia National Laboratories for supporting the work and giving me the original idea to work of record and replay.

References

[1] Asente, P. and Swick, R. with McCormack, J, **X Window System Toolkit: A Complete Programmer's Guide and Specification**, Digital Press, 1990.

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

[3] Bederson, B. B. and Hollan, J. D. *Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics*, **Proc. ACM User Interface Software and Technology** (UIST'94), 17-26.

[4] Drake, K. *X11 XTest Extension. ftp://ftp.x.org/pub/R6untarred/xc/doc/hardcopy/Xext/xtest.PS.Z*

[5] Edwards, W. K., Mynatt, E. D. and Stock, K. Access to Graphical Interfaces for Blind Users, **Interactions, 2**, 1, January 1995, 54-67.

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

[7] Libes, D. **Exploring Expect**. O'Reilly & Associates, 1995.

[8] Newmarch, J. *Using Tcl to Replay Xt Applications.* **AUUG94Conference**, Melbourne, Australia, Sept. 1994,

[9] Ousterhout, J. **Tcl and the Tk Toolkit**. Addison Wesley, 1994.

[10] Preece, J. Human Computer Interaction Addison Wesley, 1994, chapters 6 and 7.

[11] UNIX shell manual page (*man 1 sh*).

[12] UNIX pty manual page (*man 4 pty*).

[13] Zimet, M. *Extending X for Recording* (public review draft, 10 Feb 1995). *ftp://ftp.x.org/pub/R6untarred/xc/doc/hardcopy/Xext/record.PS.Z*