TkTest: Cross Platform and Remote GUI Regression Testing

A `tkreplay` Application

Clif Flynt

Noumena Corporation
9300 Fleming Rd
Dexter, MI 48130

clif@noucorp.com

Abstract

The TkReplay framework developed by Charles Crowley of University of New Mexico (crowley@cs.unm.edu) in the early 1990s has been updated to support modern widgets and MSWindows across a network use. The upgraded TkReplay has been used as the engine of TkTest, a regression test and exercise application. This paper describes how TkReplay records and replays Tk events, some of the complications and solutions in updating TkReplay to work with current widget conventions, and new features added to TkReplay to create TkTest.

Introduction

TkReplay is an application that records window events that occur while running a target application. These events can be saved and later replayed into the target application to precisely repeat a set of actions. This capability can be used to create tutorials, software demonstrations, and to exercise an application during development and testing.

Charles Crowley described TkReplay at the 4'th Tcl/Tk Workshop in 1995, and later expanded this discussion in a chapter included in **Tcl/Tk Tools**, edited by Mark Harrison, and published by O'Reilly in 1997.

The framework and sample application described by Crowley is an excellent demonstration of what can be done with TkReplay. Unfortunately, Crowley's demonstration application was the only project that used TkReplay. Perhaps because nobody had developed a killer-app around the package, it was allowed to languish while Tcl/Tk evolved and changed.

There have been many changes to Tcl and Tk since TkReplay was first developed. Back then, Tcl had been ported to several platforms but Tk was available only on Unix/X11 systems. The `send` command was available wherever Tk ran and the `socket` command had not yet been added to Tcl. Motif was the State-of-The-Art GUI library, and Windows 3.1 was considered ugly. Given this environment, TkReplay was designed to use the `send` command to communicate with the target application, it was expected that the target would accept X11 events, and an Xt/Motif style look and feel was used for the GUI.

Since then, Tk has been ported to other windowing systems including several flavors of Windows and Mac/OS, heterogeneous networks have become the norm, rather than a rarity, and MS-Windows has evolved into the intuitive GUI that users expect to see.

This paper describes:

1. how TkReplay works.
2. problems and solutions in making TkReplay work with current Tcl/Tk releases and heterogeneous networks.
3. a regression testing application built using the modified TkReplay.

TkReplay Overview

What to Record

A record and replay system can be built by capturing events at several different levels. In his 1995 paper, Crowley defined these as:

User
: Low level events that a user generates (mouse movements, button-presses, keystrokes, etc).

Lexical
: Events tied to graphical object (button or menu selections, entering or leaving a widget).

Syntactic
: Events associated with graphic commands (Redraw a screen or Scroll a window).

Semantic
: Program level events (modify a database or generate a packet).

Most current GUI regression test applications work at the User level, recording each mouse movement and keystroke, and later playing them back identically.

Two drawbacks to this approach are

1. The applications are tied to a single platform. X11 events are not one-to-one compatible to MS-Windows events, etc.

   This solution is not suitable for a multi-platform language like Tcl/Tk where we expect to run the same code with the same behavior on Windows, X11 and Mac.

2. The test scripts are fragile. Moving a window is enough to cause a test script to start a cascading failure.

Recording events at the Lexical and Syntactic levels is more robust and translates across platforms. Fortunately, Tk supports this level of monitoring and replaying events.

How To Record

The Tk `bind` command provide the ability to link window events such as mouse movement, button press and key-press events, entering and leaving windows to specific Tk widgets.

The *elegant hack* in TkReplay is to add a new binding to the existing bindings on a widget so that the widget will report the conditions when an event occurs to the TkReplay application, which can then record the event and conditions.

Like so many ideas, this is simple and elegant in concept, and not so simple in the real world.

Some of the problems Crowley solved include:

1. some widgets have internal state that must be dealt with.

   Canvas widgets have a **current** object that is selected when the mouse is over a particular object. TkReplay solves this by replacing **current** with the canvas id of graphic objects before recording the event. This allows the same object to be selected in replay, even if the exact location of the mouse is unknown.

2. some widgets allow internal graphic entities to have bindings.

   Objects in a canvas or text widget may have bindings placed on them, and the bindings may be placed on a tag that applies to several graphic objects.

   TkReplay maps all the bindings to each individual object, rather than tracking the tag names.

3. applications create new windows at run time.

   The new windows will receive new bindings. Finding all the windows at start time and adding new bindings is insufficient, the bindings must be modified as they are created.

   TkReplay renames the original `bind` command, and creates a new `bind` command that adds the TkReplay bindings to new widgets when they are created.

4. `bindtags` can cause several scripts to be evaluated for a single event.

   The `bind` command normally associates a script with and event that is connected to a particular window. However, the event can be attached to any string such as a class name (Button).

   TkReplay uses the `bindtags` command to get a list of tags that are associated with a window, and then rebinds all of the actions associated with each of the tags.

5. Namespace pollution.

   Because `TkReplay` was developed before the introduction of the `namespace` command, a naming convention was used to hide the new and renamed commands. The TkReplay commands that may be inserted into a target application include the suffix "`__rd`".

Details of Recording

The core of TkReplay is the `RebindOneWidget` procedure. Conceptually, a simple loop would add a call to the `RecordEvent` procedure as shown below.

```
   RebindOneWidget {w} {
   foreach tag [bindtags $w] {
       # rebind each event defined for this class or widget
       foreach event [bind__rd $tag] {
           bind $w $event "+RecordEvent"
       }
   }
```

The problems with this implementation include:

- if a bind script includes the `break;` command, processing the event will be terminated before reaching the `RecordEvent` command.
- some preprocessing may be required before the event is processed.

TkReplay replaces the original bind script with a call to a generic callback handler, `cb__rd`, and saves the original script in an array keyed by identifier and event (`db__rd(Bind,$tag,$event)`).

The skeleton of the RebindOneWidget code resembles this. Note that the original `bind` command has been renamed to `bind__rd` prior to evaluating this procedure.

```
proc RebindOneWidget {w} {
   global db__rd
   # ...
   # find all tags that must be rebound for this widget

   foreach tag [bindtags $w] {
       # rebind each event defined for this class or widget
       foreach event [bind__rd $tag] {

           # If binding already exists, do not rebind
           # ...

           set binding [bind__rd $tag $event]
           set db__rd(Bind,$tag,$event) $binding
           # do the rebinding
           bind__rd $tag $event \
              [format {cb__rd {%s} {%s} {%s}} $tag $event $percentFields]
       }
   }
}
```

All windows in the target application have their bindings changed from the original binding to invoking the `cb__rd` procedure.

The `cb__rd` procedure is invoked whenever an event occurs due to a user action, regardless of whether the application is recording events or not. Sections of the `cb__rd` procedure:

- decide whether to report events to the TkReplay application based on whether the target application is in Record mode.
- perform special processing based on the window class.
- report relevant events to the user.
- evaluate the original binding, and report any error conditions.

Saving and replaying events

The events are recorded as a series of Lexical and Syntactic events. These are saved internally as a Tcl command list. If the Event script is saved, it is saved as a newline delimited file in which line is a Tcl command.

The fields in the event file are:

Tcl command
: This is always the string `InsertAction`. The `InsertAction` command inserts an event into the internal event list.

delay
: Time in 1/10 second to delay before evaluating this command.

app
: The application to invoke for this event. This is either the name of the target application to receive the command, or the TkReplay application (denoted as `ThisApp`).

subscript
: The name of the action to perform. May be a `Bind,`*`windowTag`*`,`*`event`* string, `ExecTcl` (to evaluate Tcl code), `BeginComment`, `EndComment`, or begining/end of script marker.

replaceList
: A set of subscript-specific values.

  If the subscript is a `Bind` event this field contains the % field values connected with this event.

  If the subscript is an `ExecTcl` event this field contains the script to evaluate.

This is the Event Script generated when the user drives the cursor into button `.bf.save` and presses the left mouse button:

```
InsertAction {8} {DayBook} {Bind,Button,<Enter>} {{W .bf.save} {x 9} {y 16} {Args {}}}
InsertAction {0} {DayBook} {Bind,Button,<Button-1>} {{W .bf.save} {x 14} {y 12} {Args {}}}
InsertAction {0} {DayBook} {Bind,Button,<Button-Release-1>} {{W .bf.save} {x 14} {y 12} {Args {}}}
```

When an Event script file is loaded to be replayed, each line is evaluated, and the `InsertAction` command will insert the remainder of the line into a command list. This command list is used to construct `ReplayAction` commands that are sent to the target application.

The Event script show above would generate these calls to `ReplayAction`

```
ReplayAction 1 1000 Bind,Button,<Enter> {W .nt.f_2.bf.save} {x 3} {y 8}
ReplayAction 2 1000 Bind,Button,<Button-1> {W .nt.f_2.bf.save} {x 3} {y 8}
ReplayAction 3 1000 Bind,Button,<ButtonRelease-1> {W .nt.f_2.bf.save} {x 3} {y 8}
```

The first argument to `ReplayAction` is an `actionID` that is used to identify this event. The `tkreplay` application requires that target application reply to each `ReplayAction` with an `ActionEnd` containing the appropriate `actionID` before the next action is sent to the target application.

The `tkreplay` application only pays attention to the first `ActionEnd` return for a given `actionID`. This behavior allows a timeout to be set to automatically send an `ActionEnd` response whenever a `ReplayAction` command is recieved. This avoids deadlock conditions that can be caused when a widget will wait for the next event (for example, a button release during a menu motion event) before returning from the bind script.

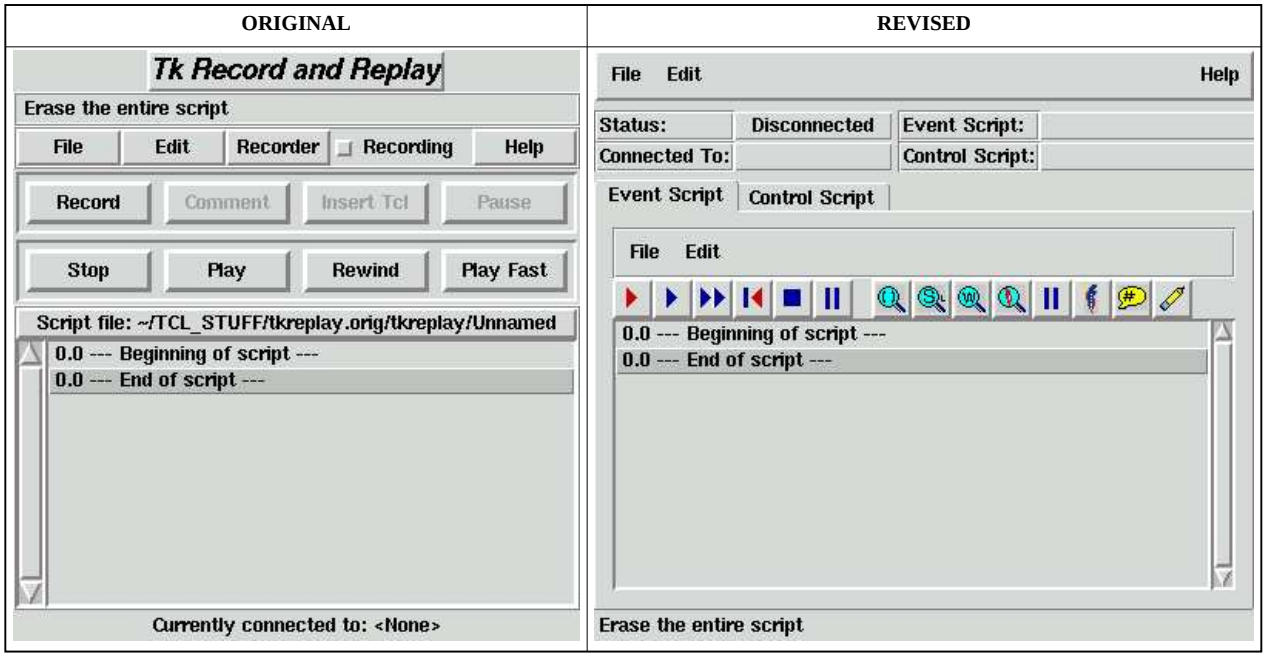
TkReplay Revitalized

The 8 year old TkReplay works surprisingly well with the basic Tk widgets and the original demonstration code. Making the package useful again required a few steps:

1. Rework the GUI.
2. Add support for new widgets (`labelframe, etc`)
3. Add support for Non-X11 widgets.
4. Remove the reliance on the `send` command.

A GUI Face-lift

The original TkReplay GUI is functional, but lacks aesthetic appeal to users accustomed to current GUI standards:



The GUI rework consisted of two steps. First modifying the appearance of the original application, and then adding GUI elements to support new features.

The modifications to the original functionality included:

- Using BWidgets for a modern appearance.
- Changing text buttons for record/replay control to task-bar buttons with common CD/DVD-Player style graphics.
- Moving status information from top to bottom of application.
- Changing menu buttons to menubar.
- Adding Labels to show current application state.

The new functionality shown in this graphic includes:

- Support for Event scripts and Control scripts.

  Event scripts are lists of recorded events to be replayed into the target application. A Control script is a list of Event Scripts. Using a Control Script makes it easier to break large tests into manageable pieces.

- Pseudo-Events

  TkReplay included support for inserting comments, pauses or Tcl commands into an Event script. I expanded this to allow the user to insert commands that will query the internal state of the target application while a recording is being made, and compare that with the state while the script is replayed later.

  This is described in more detail in the next section.

These modifications allowed me to squeeze a more runtime information into less space in the GUI, allowing slightly more space for the script to be displayed.

New Widgets

Adding support for `labelframe` and `panedwindow` was very easy. In `rebind.tcl` is this set of initialization code:

```
######################################################################
# List of widget commands.  Add your widget to this list if you want to
#    be able to record its events.
######################################################################
# USER OPTION: you can add widget creation commands to this list to get
# TkReplay to record their interactions also.
#
set WidgetCreateCommands {button canvas checkbutton entry frame label
    listbox menubutton menu message radiobutton scale scrollbar text
    toplevel labelframe panedwindow }
######################################################################
```

Simply adding the `labelframe` and `panedwindow` to the list was sufficient to allow make these to be recognized as widget creation commands.

Other widgets required modifications to the TkReplay base code to handle their behavior.

For example, the megawidgets that existed with Tk 4.0 did not use the `after idle` command. The current `tk_getOpenFile` and `tk_getSaveFile` megawidgets use this command to update the selection window.

Thus, this simple code:

```
if {$db__rd(Replaying)} {
   # do not do "after"s while replaying
   return
 }
```

became this slightly less simple code:

```
if {$db__rd(Replaying)} {
   # do not do "after"s while replaying
   #  -- Except for idle - this is necessary for the
   #   file dialog that uses "after idle" to update the
   #   selection window.
   if {[string equal $subcommand idle]} {
      if {$args != {}} {
          uplevel 1 after__rd $subcommand $args
      }
   }
   return
 }
```

Modal widgets may use the `vwait` command to halt execution until the user performs an action. This causes a lock condition, when the target application is waiting for input and the TkReplay application is waiting for a response from the target application before it sends the input that the target application is waiting for.

A new `vwait` command was created that will send the response after a timeout, then invoke the real `vwait` (`vwait__rd`) procedure to wait for the variable to change.

```
proc vwait {varName} {
    global db__rd

    if {$db__rd(Replaying)} {
        # After the timeout period, send a reply back
        # to the application then vwait.
        after__rd $db__rd(timeout) [list catch \
            [list tkrsend $db__rd(ReplayApp) \
                [list ActionEnd $db__rd(actionID) timeout]]]
    }
    uplevel 1 vwait__rd $varName
}
```

This causes the TkReplay application to receive two `ActionEnd` events for some actions. Only the first event will be processed, and others are silently discarded.

Opening Windows

The two primary difficulties with using TkReplay on Windows-based computers are the lack of a `send` command, and the use of native Windows widgets that don't match the behavior of the equivalent X11 widget.

The `send` problem was solved by Crowley when he added support for socket communication as well as the `send` command.

The socket code used with TkReplay duplicates the behavior of the `send` command. This added more complexity than my application required, and I removed much of this code and replaced it with a simpler socket protocol which is discussed later.

The Windows `menu` command required a great deal of widget specific handling. The normal process of using `bindtags` to find the bindings on a widget and modifying them did not work with Windows menus. The menus would not display, cascading menus didn't cascade, button clicks were ignored, etc.

The problem is that once the control is given to the native Windows Menu widget, the Tcl event loop is loses control. This causes the deadlock condition described above, and makes the events normally associated with a menu irrelevant.

Forcing the `ActionDone` message to be sent after a short timeout solves the deadlock condition, but is not sufficient to make menus behave under windows, since the X11 Window events are not recognized by the Windows menu.

The solution was to add special bindings for the menu commands to invoke menu subcommands including *`$menuName post`*, *`$menuName invoke`*, and *`$menuName activate`* as required.

One variant of Windows and Tk that was tested defined duplicate names for the menu windows. The duplicates are the same as the original with a "#" character added. Binding to both names caused duplicate events to be evaluated. Adding a check for the # solved this problem.

Here is the modified section of code that provides special-case windows menu bindings:

```
if {[string equal Menu [winfo class $w]] &&
    ([string first # $w] < 1)} {
    bind $w  "WinMenuMotion $w %W %x %y"
    bind $w  "WinMenuRelease $w %W %x %y"
    bind $w  "WinMenuPress $w %W %x %y"
    set db__rd($w.active) none
  }
} else {
# ... perform normal bindtags/bind commands
}
```

This does not allow menu events to be recorded from a Windows platform, but does allow a an event script recorded on an X11 platform to be replayed on an application running on a Windows platform.

The solution to the File Selection dialogs is a less transparent kludge. On Windows platforms, Tk uses the native Windows File Selection widgets. Like the native menus, these are ill inclined to play well with others. Fortunately, on X11 there is no native file selector, so Tk comes equipped with a pure Tcl file selector that provides all the features of the windows selector. The pure-Tcl file selector uses the `::tk::dialog::file` command which can be used to overwrite the Windows specific widgets by adding code like this to the main application:

```
    proc ::tk_getOpenFile {args} {
        if {$::tk_strictMotif} {
            return [eval tk::MotifFDialog open $args]
        } else {
            return [eval ::tk::dialog::file:: open $args]
        }
    }

    proc ::tk_getSaveFile {args} {
        if {$::tk_strictMotif} {
            return [eval tk::MotifFDialog save $args]
        } else {
            return [eval ::tk::dialog::file:: save $args]
        }
    }
```

Rescinding `send`

The original TkReplay relied on the Tk `send` command to communicate with the target application. The good side of this is that it allowed an application to be tested without any instrumentation. The bad sides are that it doesn't allow for applications to be tested across a network and `send` is not supported on all windowing systems.

The TkReplay design includes a software layer over the actual communications protocol which allowed the use of sockets as an option. This which made it relatively easy to remove the residual support for `send` and replace it with a new socket protocol.

The downside of this is that a target application must now be instrumented in order to have events recorded or replayed.

The model I used for the revised TkReplay is to make the TkReplay application a server, and the target application a client. The client must source one file with the socket protocol commands, and the rest of the instrumentation is handled by the TkReplay server.

This is the minimal boilerplate I add for an application:

```
# Invoke as "wish main.tcl -testHost 127.0.0.1" for testing
if {![string equal "" $StateArray(testHost)]} {
  # INVOKE tktest as: wish tktest.tcl

  source socksend.tcl
  sockappsetup tkreplay.tcl 3010 $StateArray(testHost)

  # Include tk_getSaveFile/tk_getOpenFile stubs if necessary.
}
```

The communication protocol is simple, and duplicates half of the `send` functionality. Tcl commands are sent from client or server, and evaluated by the receiver when a complete command is received. The results of a command are not explicitly returned.

When a client connects, the first thing the TkReplay server does is to send the client all required instrumentation code (rebinding procedures,etc) via the socket. One advantage to this technique is that during TkReplay development, changes to files in the TkReplay directory are automatically reflected in the target application. Only `socksend.tcl` needs to be copied to the target application directory if it is modified.

TkTest: A TkReplay Application

My primary goal for working with TkReplay was not to play with interesting ideas and see how far the envelope could be pushed, but to perform regression testing on applications that were already on tight deadlines.

Most software engineers test the changes they make to an application by running the application and observing the behavior in the area they are modifying. Depending on the complexity of the application and modification, this can be a few program interactions done a few times, or several hundred program interactions repeated hundreds of times over days.

A human will not reliably reproduce the same actions time after time, and not-quite repeating a set of actions can conceal the effects of a change and lead the developer down the wrong path for a while.

The ability to replay window events solves this problem. It allows the developer to replay identical interactions and observe the changes caused by code modifications without having to memorize (and be certain of duplicating exactly) a sequence of events.

When testing new code, it’s too common to skip verifying that the existing code still works as it did originally. It is all too common that the fix for one bug introduces a new one. This causes projects get bogged down in endless cycles of fixing one bug while introducing another.

Again, the ability to replay events allows developers to create Event scripts that will exercise the entire application, and confirm that everything still works without crashing.

Being able to replay events repeatably is useful for exercising a target program but it is not sufficient for rigorous testing, since it does not monitor the state of the target application. For example, a tired developer might not notice that a button is missing, an unused menu has lost an entry, or that clicking a button does not actually perform a function.

TkTest is an extension of the TkReplay application that includes facilities for reporting the state of the target application while recording an Event Script and saving that state for comparison while replaying a set of events.

This facility allows the developer to create scripts which both exercise and test an application, confirming that user actions cause the expected results. The Event Script with expected results can be saved for future use. The script can be used during a development effort repeat the same tests, confirming that a code change did or did not affect the behavior, and can be used later as a regression test to confirm that new

modifications do not affect existing code.

In practice, using TkTest to test an application requires a few steps to generate an Event Script:

1. Add instrumentation code to source `socksend.tcl` and initialize the connection (as shown above).
2. Press the red *Record* button on the TkReplay **Event Script** tab.
3. Perform some actions in the target application.
4. Click a Magnifying Glass button to record the state of the target application.
5. Optionally, continue using the target application or inserting more tests.
6. Stop the recording with the *Stop* button on the **Event Script** tab.
7. Optionally, compress motion events.
8. Save the script for future use.
9. Optionally add this Event script to a Control Script for lengthy tests.

After modifying the target application, the test can be rerun by starting the target application (without restarting TkReplay), waiting for the connection to be complete, and pressing the blue *Play* or *FastPlay* button on the control bar.

The recording, replay and introspection functions are made available on a control bar in the Event Script window. The control bar looks like this:

The buttons on the left are the recorder controls, and the buttons on the right modify the recorded script.

The recorder control buttons are:


Start recording events. Events will be displayed in the script window as they are recorded.


Play events. The event being replayed is highlighted in the script window.


Play events quickly. TkReplay records the time of events, and by default will play back events with the same timing (or slightly slower) than they were recorded. This button disregards the timing information and plays back as quickly as possible.


Rewind the event pointer to the top of the script window.


Stop playing this Event Script.


Pause the event playback.

The Introspection and Test control buttons are:

 Records and checks the contents of a global array in the target application.

 Records and checks the results of an SQLite database query in the target application.

 Records and checks the contents of window hierarchy in the target application.

 Records and checks the return value of a Tcl script evaluated in the target application.

 Add a "Pause" to the script

 Add a command to be evaluated in the TkReplay application.

 Add a comment to the script.

 Erase the currently displayed script.

The introspection and test commands all follow the same pattern. A command is sent to the target application which returns the results to the server. These results are attached to the command that generated them and the resulting retrieve-and-compare command is placed in the TkReplay event

For example, while recording data, the user might press the SQL query button, to take a snapshot of the database. This will send a command to the target application to execute a database query, and return the results to TkReplay. The TkReplay application will then fabricate an event to repeat this query during replay and compare the new results with the results obtained during the recording.

This line would be inserted into the Event script:

```
InsertAction 0 DayBook ExecTcl \
  {{CheckReturn {db eval {SELECT name FROM project}} \
     {InstantDinner BigRocket {Rolling Rock}}} 1}
```

This causes the `CheckReturn` procedure to be invoked during replay. The `CheckReturn` procedure sends the `db eval {SELECT name FROM project}` command to the target application, and then checks that the return string matches `InstantDinner BigRocket {Rolling Rock}`.

The code that implements this process is:

```
#################################################################
# proc CheckReturn {cmd expect}--
#   Check the return from a command and report match/nomatch
# Arguments
#   cmd         The script to be evaluated in the target
#   expect      The expected return string
# Results
#   A command is evaluated in the target application, which may
#   change the target application state.
#   A message is generated to the user.
#
proc CheckReturn {cmd expect} {
    global ReplayData

    set rtn [getReply "$cmd"]
```

```
    if {[string equal $expect $rtn]} {
      set msg "Cmd Check OK ($ReplayData(ScriptFileName)): $cmd"
    } else {
      set misMatch [stringDiff $rtn $expect]
      set msg "Cmd Fail ($ReplayData(ScriptFileName)): $cmd\n  $misMatch"
    }
    MsgToUser "$msg" high
}
```

The socket communication protocol of sending Tcl commands to be evaluated by the receiver simplifies most communication, but makes sending non-command data strings a bit tricky. The solution I used was to have the sending program send a `set globalVariable "string"` command to the other application, which uses vwait to hold until the data is recieved.

The `getReply` procedure is implemented as:

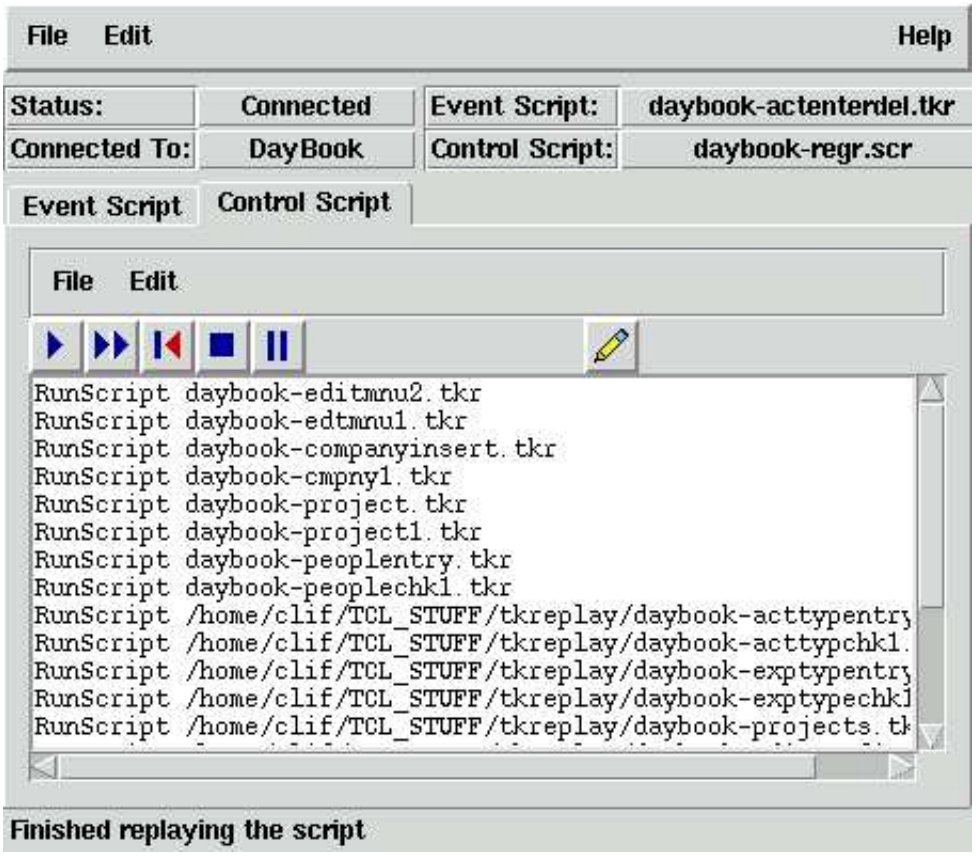
```
#####################################################################
# proc getReply {cmd}--
#    Send a command to the target application and wait for
#    the reply.
# Arguments
#   cmd         The script to send to the other application.
#
# Results
#   A script is evaluated in the other application which may change
#   change that application's state.
#   Global variable ReplayData(Reply) is modified.
#
proc getReply {cmd} {
  global ReplayData

  tkrsend $ReplayData(ConnectedApps) \
    "catch {$cmd} rpl; \
    socksend  tkreplay.tcl \"set ReplayData(Reply) \[list \$rpl]\""
  vwait ReplayData(Reply)
  return $ReplayData(Reply)
}
```

Control Scripts

Real-world applications have many sets of user interactions. Developing a single Event Script to test an entire application would be long, slow and unpleasant.

TkReplay included support for calling one script from another. I expanded this functionality into a separate tab as shown below.



The task-bars are simplified for this functionality. The *Recorder* does not include a *record* button, since recording events is done on the `Event Script` window. The only edit button ( ) will insert a new script.

The text window shows the list of Event Scripts that will be played as when this script is played. Each of these scripts is loaded and evaluated in turn.

Current Status and Future Work

The current status of TkTest is a strong Alpha release. It has been used successfully to run small Event Scripts while developing a subsystem on several projects, and as a complete regression task on one another application.

This does not imply that TkTest is complete or fully debugged.

Future development includes:

- Make use of namespaces.

  The current code uses procedure naming conventions to avoid collisions. Using namespaces would allow the state variables to be taken out of global scope, and would hide the procedure names.

- Remove unused code.

  There are some sections of the TkReplay code that are no longer used.

- Improve megawidget and Windows widget support.

  The current code works for the known widgets, but generic solutions to working with native Windows widgets would be much better.

- User Interface Improvements.

  The best that can be said for much of the current GUI is that it works. More intuitive and informative error returns would make it easier to interpret TkTest results.

- Add facility for inserting events into an Event Script.

  The current *record* behavior is to start at the top of the list and insert new events. When editing and updating scripts it would be useful to be able to place a *current* pointer into the list and start inserting new events at that point.

- Simplify Event Script editing.

  The Event Script is a flat ASCII file which can be edited with any text editor, given that the user understands the file format and is careful.

  The Event Script format is sufficiently regular and well defined that a special purpose editor could be made that would guarantee conformance to the expected formats.

- Further socket protocol streamlining.

  The current version of the socket protocol only allows communication between the TkReplay server and a single target application client.

  The use of application names in establishing the connection is only required because of parts of the legacy code. Future work will either remove these vestiges, or allow the TkTest master to control (and perhaps relay events between) multiple clients.

Conclusion

Computer applications can be divided into two categories. Those that are unused, and those that need more work.

The TkTest application is being used to test and exercise applications on both Linux and Windows, locally and across a network. Therefore, it needs more work.

This paper, and a TkTest snapshot are available at http://www.mod3.net/~clif