Regression testing GUIs
New features for tktest

Clif Flynt
Noumena Corporation,
8888 Black Pine Ln,
Whitmore Lake, MI 48189,
`http://www.noucorp.com`
`clif at noucorp dot com`

September 27, 2015

Abstract

Repeatable, interactive testing is essential to develop even a simple application. During development, a few tests that exercise the functionality being developed while the developer monitors the results are sufficient.

Complex applications and applications in the maintenance/extension phase of their existence require too many tests to be manually monitored. All portions of the application must be exercised to avoid the dreaded ”removed one bug, added two new ones” phenomenon.

The tktest application has been expanded to support automated regression testing. These expansions include facilities for creating a log file for each test, tools to examine the log files, improved facilities for recreating tests, and new test modes to reduce false negatives.

1 Introduction

1.1 History

In 1995, Charles Crowley described the TkReplay application at the 4th Tcl/Tk Workshop. This application allows the user to record and replay events to a Tk application using the Tk `send` command.

TkReplay is a useful tool for exercising an application during development, but by the early 2000s the GUI was outdated, and there was a need for more functionality.

In 2004, Clif Flynt described expanding and updating the TkReplay program into the `tktest` application at the Tcl/Tk conference.

The `tktest` application supported all the features of the original TkReplay project and added:

- modern BWidget style GUI.
- support Windows and MacOS.
- communication via sockets instead of `send`.
- support for platform-native widgets.
- introspection into the target application.

The 2004 version of the `tktest` application proved useful for exercising code. The new support for examining the application-under-test's internal structure made it particularly useful during the phase in which a developer is busy adding new features. The ability to introspect into window hierarchies, data structures and attached databases provides tools for confirming that new code is not breaking old code.

Between 2004 and 2015 the `tktest` application has been used with several applications. It has been used in the development phase of for many projects, and is in post-development use to demonstrate filling out fields in a help system and to generate customized distribution kits by querying a database and running the Admin GUI to add users and clients.

The application has been extended to work with `ttk::` widgets as well as traditional widgets including introspecting into notebooks and paned windows. There is improved support for overriding platform-native widgets.

1.2 Current Status

The `tktest` application has proven useful during the development phase of a project. Being able to run a complex set of interactions with a single button click not only saves time but frees the developer to concentrate on solving a problem instead of remembering how to reproduce the problem.

As a project moves from development to deployment to maintenance, the testing needs change. Tktest works well for the focused interactive tests in the development phase of a project. When a project moves to the maintenance stage of its life, it needs fuller automated test coverage.

In order to make `tktest` useful for regression testing a number of new features were added to tktest and new helper applications were created.

This paper describes how to use `tktest` in interactive mode to develop tests, how to collect those tests into a regression suite, how to run such a suite and examine the results and finally the decisions and design that went into creating the new `tktest` regression framework.

2 TkTest design

The backbones of TkReplay and the `tktest` application are Tcl's ability to rename functions, perform introspection on a running program and the object-oriented nature of the Tk widgets.

When `tktest` starts it opens a server side socket and waits.

When the target application starts, it opens a client side socket to the `tktest` server. The server then sends commands to be evaluated in the client. These Tcl commands rename many of Tk's standard procedures and create new procedures to instrument the application being tested. The server then invokes the procedure to step through each widget in the application and extend the bindings for windows events like `<Enter>`, `<Leave>` the mouse buttons, etc.

The new widget bindings add a call to send event information to the `tktest` server whenever an X11 event occurs in the client. When the server is in *Record* mode, it saves these actions for later replay.

New widgets automatically receive the extra bindings courtesy of renamed and overridden widget and `proc` procedures.

The server can send other Tcl commands to the client during recording and replay. The introspection buttons send commands like `array get $arrayName` and record the return from evaluating that command. These commands allow the server to introspect into a running client to confirm that internal variables have the expected values, etc.

The ability to send commands to the client can also be used to test the returns from internal procedures or perform actions that can't be easily automated via the GUI.

3 TkTest in Interactive Mode

Two common patterns during development are:

- use the `tktest` application to drive the application being developed to a state where further manual testing can be done.
- : use `tktest` to create a script that generates an error condition that's being debugged.

Both patterns use this set of steps:

1. Instrument the application to be tested. In order to interact with an application, the application must be instrumented. This can be done with two lines of code:

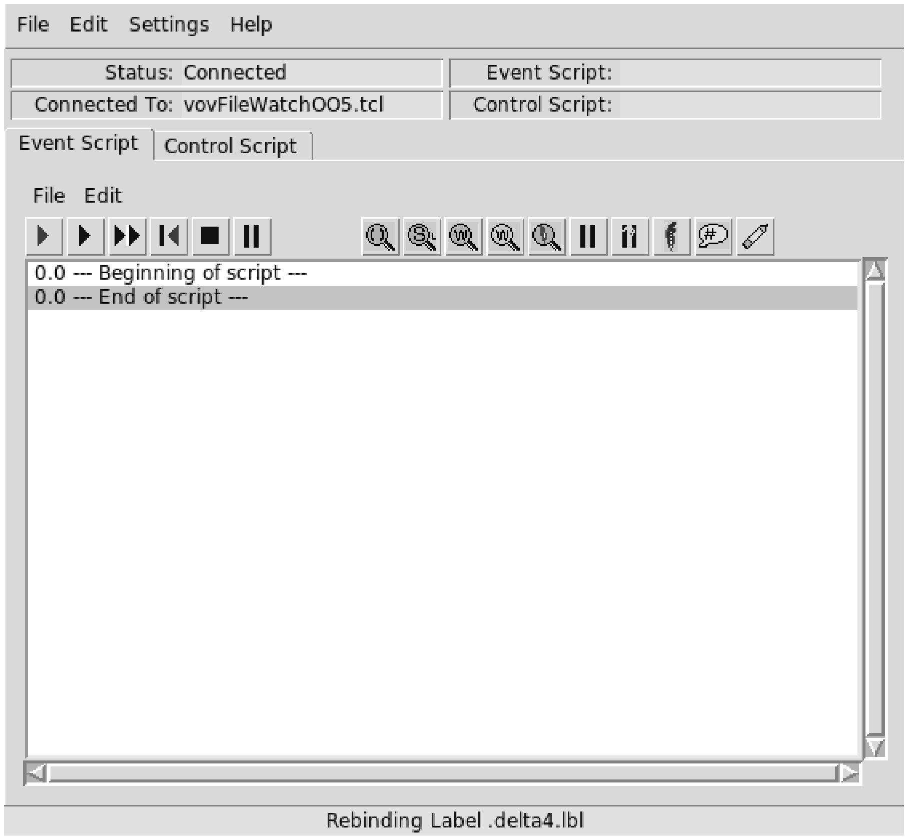
   ```
   source socksend.tcl
   sockappsetup tktest.tcl 3010  127.0.0.1
   ```

   An application that includes these two lines will pause at the `sockappsetup` call until a `tktest` application accepts the socket connection.

2. Start `tktest`

   When `tktest` starts it opens a server-side socket to communicate with the application under test. The client initiates the connection and `tktest` sends commands to fully instrument the client application.

After the connection is completed the tktest application will resemble this:



Status: Connected

Connected To: vovFileWatchOO5.tcl

Event Script:

Control Script:

```
0.0 --- Beginning of script ---
0.0 --- End of script ---
```

Rebinding Label .delta4.lbl

Figure 1: tktest screen

3. Click leftmost arrow to put `TkTest` into *Record* mode.

   The *Record* button is the red action button in the Event Script tab.

4. Perform actions you want to be repeated in the test application

   Every event in the test application will be sent to `TkTest` where it is timestamped and saved.

5. Stop recording in the `TkTest` application

   The action button with a black square stops recording. The `TkTest` application can now rewind and replay a set of actions.

6. Restart the test application and rerun the script

The client application can be stopped and restarted without restarting the `TkTest` application. This allows you to edit, restart, use `TkTest` to drive the application to a known state and either manually do more testing, or add tests to be run automatically by `TkTest`.

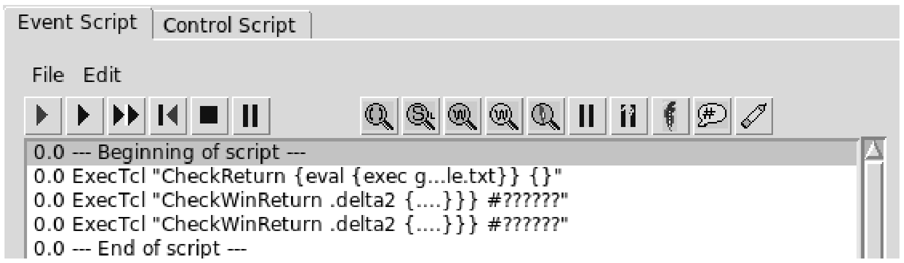
The event script can be rerun immediately by clicking the *Rewind* and *Play* buttons, or saved for future use via the *File/Save* menu item.

3.1 TkTest Overview

The `tktest` GUI consists of three sections:

- the top menu bar.
- status display.
- tab notebook.

In practice, you spend the bulk of your time dealing with the controls in the notebook panels. The menus and buttons on those panels support loading and saving event scripts, recording and replaying the event scripts and inserting introspection into a test script.



```
0.0 --- Beginning of script ---
0.0 ExecTcl "CheckReturn {eval {exec g...le.txt}} {}"
0.0 ExecTcl "CheckWinReturn .delta2 {....}}} #??????"
0.0 ExecTcl "CheckWinReturn .delta2 {....}}} #??????"
0.0 --- End of script ---
```

Figure 2: Event Script window

The buttons shown below control the record and playback features of `tktest`.

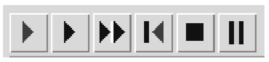


Figure 3: Record/Playback buttons

Record/Play Buttons

▶	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 recording and return to playback mode or stop playing the current Event Script.
❚❚	Pause the event playback.

While recording events in the test application introspection events can be added with the test buttons shown below.

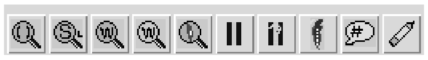


Figure 4: Introspection buttons

The Introspection and Test control buttons are:

Introspection Buttons

Records and checks the contents of an 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. There are two forms of this button, one that gathers all information about the window and one that rejects the background color.

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. This may be to wait for an event in the client to occur, allowing the coordination when events take an indeterminate length of time.

Add a comment to the script.

Erase the currently displayed script.

3.2 Complex Test Sequences

A single event script is useful when developing an application and targetting a localized area of code.

The limitation to simple event scripts is that they can become long. The longer an event script becomes, the more it becomes single-use and fragile.

For example, consider adding a "Save Configuration" feature to a piece of code. The event script will need to perform actions like:

- Open configuraton menu.
- Select desired settings.
- Close configuration menu.
- Open Save Configuration dialog.
- Enter save parameters.
- Click SAVE button.

This script works fine until you add a new configuration option and want to confirm that it's saving properly. You'd need to repeat a long set of actions to build a new Event Script. That requires repeating a lot of actions to rebuild the Event Script with a few new actions in the middle.

Event scripts can be combined with a *Control Script*. In the case above we might make four event scripts:

- Open configuraton menu.
- Select desired settings.
- Close configuration menu.
- Save Configuration

When it's time to add a new configuration option, we simply insert it in the middle:

- Open configuraton menu.
- Select desired settings.
- SET NEW CONFIGURATION OPTION
- Close configuration menu.
- Save Configuration

Other tests might just use the configuration setup Event Scripts to put the test application into a known state or use the "Open configuration menu" Event Script with a different set of parameters followed by the "Close configuration menu".

Adding the ability to mix and match Event Scripts is another step toward making `tktest` useful for regression tests.

4 Regression Testing

In theory, there is no difference between theory and practice.
In practice, however...

In theory, it would be simple to use the event scripts that were used during development testing to do regression testing.

In practice, this doesn't quite work.

Loading and running a single test while you're developing an application is a time saver. Loading and running a dozen tests and examining the output is too slow and tedious ever to happen.

Running regression tests needs to be automated; there must be a summary report generated; the results must be saved for later examination and there must be tools to facilitate that examination.

The first obvious modifications to `tktest` were command line arguments to let a test be automated.

Command Line Arguments

-appNames	Define the application to exec
-script	Define the event script to load
-playScript	Define play speed for automated runs [normal fast fastonce]
-recordFile	Define the file to record the results
-strict	Should windows checks be strict or loose (default 1)
-exitChild	Send exit cmd to child after run (default 0)

These command line arguments let a set of event scripts be automated with a shell script like

```
wish tktest.tcl -appNames myTestApp.tcl -script test1.tkr \
    -playScript fastonce -recordFile test1.txt -exitChild 1
wish tktest.tcl -appNames myTestApp.tcl -script test2.tkr \
    -playScript fastonce -recordFile test2.txt -exitChild 1
```

The above commands will start `tktest` and then `tktest` will exec the application being tested, load and run the event script, save the results and finally send an "exit" command to the test application before exiting itself.

Again, this seems pretty straightforward and simple until it reaches the real world.

A run-test script is adequate for a few tests, but:

- keeping the shell script file up to date as new tests are added is tedious.
- grummaging through the individual report files is even more tedious.
- some tests require pre-test setup or post-test cleanup.

These requirements drove the need for a more powerful control script and a file hierarchy for the tests and support files.

The `tktest` application is evolving as it meets more reality.

The current design is a top level folder for all the tests related to a project. Within that folder are individual folders for each test.

The current file structure resembles this:

- Master_Folder
    - generic support files
    - *TestName*_Folder
        - preTest.tcl: pre-test file setup and define params()
        - postTest.tcl: test cleanup
        - *TestName*.scr: Test script for this test.
        - *scriptName1*.tkr: Individual event scripts to be run.
        - *scriptName2*.tkr: Individual event scripts to be run.

The `runRegressionTests.tcl` script accepts the master folder as a command line argument and then steps into each of the child folders. It sources the `preTest.tcl` script to perform any pre-test setup and to define the `param` associative array that defines values used by the test.

The `preTest.tcl` file looks like this:

```
file copy -force ../shortSample.txt /tmp/sample.txt
array set params {
  appNames {{./test2.tcl}}
  tktest ../../../tktest/tktest.tcl
  wish   /usr/local/bin/wish
}
```

The `runRegressionsTest.tcl` script eventually starts `tktest.tcl` with this command line:

```
  set rtn [exec $testParams(wish) $testParams(tktest) \
    -script $root.scr \
    -recordFile ${root}Results.txt \
    -playScript $testParams(playScript) \
    -exitChild $testParams(exitChild) \
    -appNames $testParams(appNames)]
```

The default values for `playScript` and `exitChild` cause `tktest` to run once quickly, send the `exit` command to the child and then exit itself.

These values can be changed on the command line to `runRegressionTests.tcl` to cause `tktest` and the application being tested to remain active during debugging.

When the regressions are complete, there is a results file in each of the child folders and a summary file in the master folder. The summary file is named with a time/date stamp to make it easy to track results. The individual results files are overwritten after each run.

5 Analysis of Results

Running tests is all just fun and games until one fails.

A failure may be because:

- a test is miswritten and is checking for values that are not constant, for example, a label containing the current date
- because the functionality was changed.
- because the GUI has been broken.
- because the GUI has been upgraded in a manner that breaks the test.

The `tktest` application generates a report file describing the differences but that file may be several hundred thousands of bytes with only one character of difference.

The obvious solution to this was to use `tkdiff` to highlight the difference, but that only works for simple problems. When the difference is a single character in a complex window hierarchy, you need a context sensitive diff, not a generic diff.

Tracking down the discrepancies needed a new application. The new application is cleverly named `resultsViewer.tcl`. It provides two views of the results file.

5.1 Example of use

Here's a simple wish application that puts up two labels, one of which will display a value from the command line.

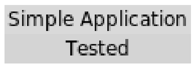
```
source socksend/socksend.tcl
sockappsetup tktest.tcl 3010  127.0.0.1

pack [label .title -text "Simple Application"]
pack [label .l -text $argv]
```

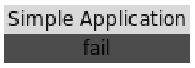
This test application can be run with `tktest` using the command shown below. This command will run the `singleDiff.tcl` file and providing ”fail” as the command line arg. It will play the Event Script with no time delays once. After running the script both the `tktest` application and the client will exit.

```
wish tktest.tcl -appNames "{./singleDiff.tcl fail}" \
    -playScript fastonce \
    -exitChild 1 \
    -script singleDiffTest.tkr \
    -recordFile singleDiff.txt
```

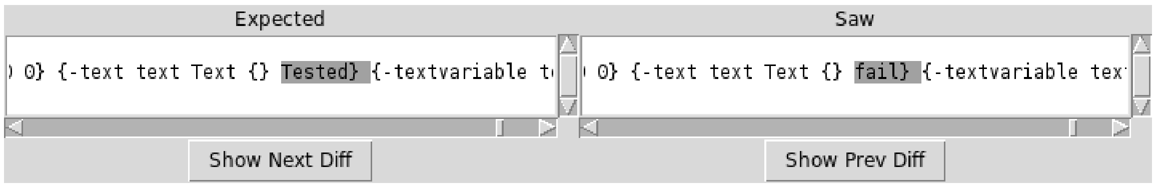
The `resultsViewer.tcl` application can be used to examine the `singleDiff.txt` report file. The `resultsViewer` application will create three toplevel windows. One to show the expected GUI with discrepancies highlighted, one to show what was actually seen, again with discrepancies highlighted, and finally a `tkdiff` style difference between the two text files.



Expected display



Observed display



```
) 0} {-text text Text {} Tested} {-textvariable t
 0} {-text text Text {} fail} {-textvariable tex
```

Figure 5: Textual differences

5.2 Making better tests

The *Edit* menu on the `Event Scripts` panel includes several items to facilitate updating existing tests or creating better tests. The first elements are actions that work on an entire script while the last section has actions that modify a single event.

The main display in the `tktest` application is a listbox that supports selecting a single line. Each event is reduced to a single line in this display, even though it may be several hundred lines long. Once a line is selected, the `Delete` and `Edit` options will process that line.

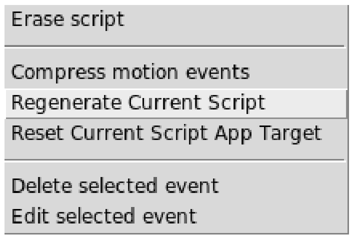


Figure 6: Event scripts Edit menu

5.2.1 Too many events

One of the first things you'll notice when you run `tktest` is that there are a lot of events generated. Many of these are superfluous movement events. There is nothing wrong with these events, except that they clutter up the screen and take time to evaluate.

The `Compress motion events` menu item removes all the intermediate events and replaces them with a single movement from the starting location to the end location. This frequently cuts an event script to a quarter of the original size.

5.2.2 New values for an old test

If you've done a minor cleanup of a GUI, for example replacing one font with another or remembering to capitalize all the titles, you won't want to rebuild a long test script from scratch. The `Regenerate Current Script` will rerun the actions and generate a new script with the current values returned from the introspection calls.

This action will save the new script under a name provided by the user.

5.2.3 Resetting the application target

An event script knows the name of the application being tested. In theory, `tktest` could interact with multiple targets. (In practice, this has not been implemented.)

If you change the name of the application under test, you will need to retarget the script. This can be done with the `Reset Current Script App Target`.

This action happens in-place within the `tktest` application, which allows the user to load a script, connect from a test application, retarget and run the Event Script.

5.2.4 Removing an event

People are only human. Sometimes you add an event that you really didn't want to use after all. The `Delete selected event` menu element will remove the selected event.

5.2.5 Modifying an event

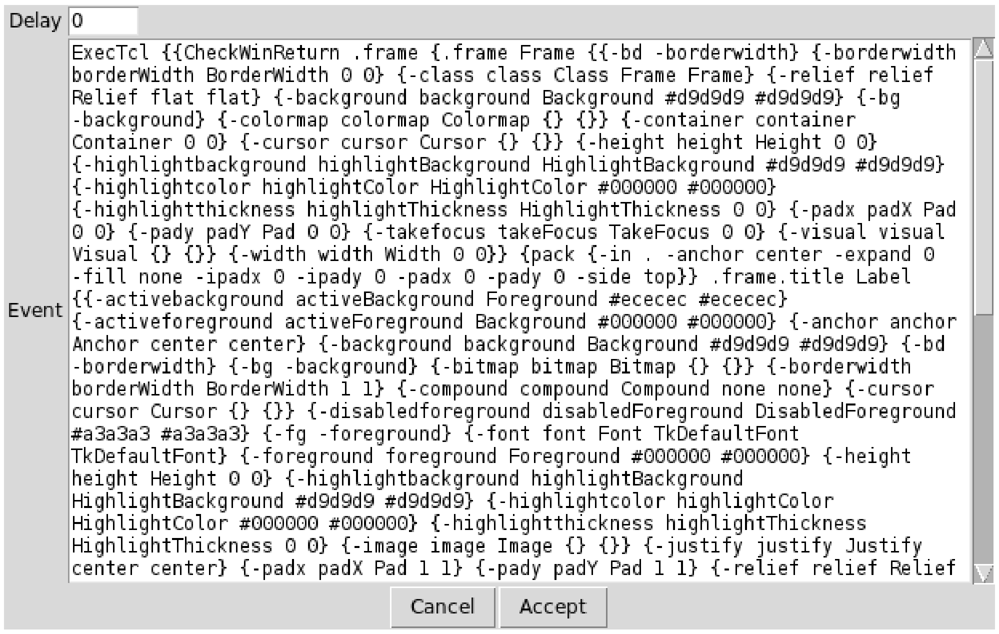
The default behavior of `tktest` is to collect data from a gold run of your application and confirm that it exactly matches the current run. This is is a simple test, but not always best. In some cases values need to be discarded before performing the tests, in other cases a vague match is better than precise, etc.

For example, you can check the contents of an array with a single command. This is much faster than entering a command for each element in the array. However, if one field in the array is a timestamp, it's guaranteed that the test will fail.

The `tktest` application uses `string match` to compare the expected returns to the actual value returned. This allows the expected pattern to include glob wildcards instead of exact text.

The Event Scripts are plain ascii text files. They can be modified with your favorite text editor, or they can be edited within `tktest`.

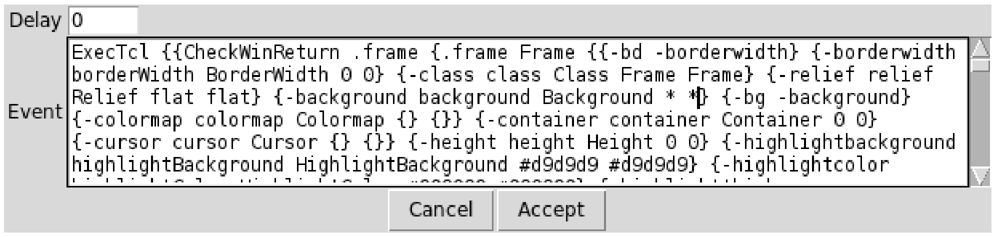
Note that editing an event is fraught with danger. The event may generate thousands of bytes of introspection data and the format is critical. Experienced Tcl programmers should have little trouble with the format–it is simply Tcl, but be careful about matching braces, spaces, etc.



```
ExecTcl {{CheckWinReturn .frame {.frame Frame {{-bd -borderwidth} {-borderwidth
borderWidth BorderWidth 0 0} {-class class Class Frame Frame} {-relief relief
Relief flat flat} {-background background Background #d9d9d9 #d9d9d9} {-bg
-background} {-colormap colormap Colormap {} {}} {-container container
Container 0 0} {-cursor cursor Cursor {} {}} {-height height Height 0 0}
{-highlightbackground highlightBackground HighlightBackground #d9d9d9 #d9d9d9}
{-highlightcolor highlightColor HighlightColor #000000 #000000}
{-highlightthickness highlightThickness HighlightThickness 0 0} {-padx padX Pad
0 0} {-pady padY Pad 0 0} {-takefocus takeFocus TakeFocus 0 0} {-visual visual
Visual {} {}} {-width width Width 0 0}} {pack {-in . -anchor center -expand 0
-fill none -ipadx 0 -ipady 0 -padx 0 -pady 0 -side top}} .frame.title Label
{{-activebackground activeBackground Foreground #ececec #ececec}
{-activeforeground activeForeground Background #000000 #000000} {-anchor anchor
Anchor center center} {-background background Background #d9d9d9 #d9d9d9} {-bd
-borderwidth} {-bg -background} {-bitmap bitmap Bitmap {} {}} {-borderwidth
borderWidth BorderWidth 1 1} {-compound compound Compound none none} {-cursor
cursor Cursor {} {}} {-disabledforeground disabledForeground DisabledForeground
#a3a3a3 #a3a3a3} {-fg -foreground} {-font font Font TkDefaultFont
TkDefaultFont} {-foreground foreground Foreground #000000 #000000} {-height
height Height 0 0} {-highlightbackground highlightBackground
HighlightBackground #d9d9d9 #d9d9d9} {-highlightcolor highlightColor
HighlightColor #000000 #000000} {-highlightthickness highlightThickness
HighlightThickness 0 0} {-image image Image {} {}} {-justify justify Justify
center center} {-padx padX Pad 1 1} {-pady padY Pad 1 1} {-relief relief Relief
```

Figure 7: Event scripts Edit menu

The image below shows the edit screen for the single `CheckWinReturn` test of the two window `Simple Application` described above.



```
ExecTcl {{CheckWinReturn .frame {.frame Frame {{-bd -borderwidth} {-borderwidth
borderWidth BorderWidth 0 0} {-class class Class Frame Frame} {-relief relief
Relief flat flat} {-background background Background * *} {-bg -background}
{-colormap colormap Colormap {} {}} {-container container Container 0 0}
{-cursor cursor Cursor {} {}} {-height height Height 0 0} {-highlightbackground
highlightBackground HighlightBackground #d9d9d9 #d9d9d9} {-highlightcolor
```

Figure 8: Event scripts Edit menu

5.3 Loose and strict tests

The default behavior for `tktest` is to compare all available data.

This may be superfluous. Some windows values may not affect the actual display or may not matter to the behavior of the program.

While constructing a test the user can select the color or no-color option for window introspection. In the no-color mode, background colors are disregarded.

At runtime, the user can select strict or loose window checking. In loose mode, these elements are discarded from the expected and observed returns before comparisons are done:

```
-sticky -column -columnspan -in -ipadx -ipady -padx -pady -row
-rowspan -browse -width -height -anchor -borderwidth -cursor
-compound -font -justify -orient -padx -pady -setgrid -takefocus
-textvariable -selectmode
```

By default `tktest` uses the strict mode.

6 Future work

There are two types of computer programs: ones that are not being used and ones that need to be modified.

The `tktest` application is firmly in the second camp. Every time it gets used with a new project I find new features to add or old features that need to be modified.

Items on my ToDo list include:

- Many options including `-strict` and `-usecolor` are currently hard-coded. These should be customizable from a configuration file and command line.
- Customizing an Event Script currently requires editing the .tkr script. This should made easier.
- Analyzing the results is an area that always needs more work. The current tool, `resultsViewer.tcl`, is better than previous tools, but it still difficult to determine the exact reason a test has failed. In particular, finding the correct widget in a nested set of frames, panels and notebook tabs is difficult.
- After 20 years of hacking and last-minute fixes sections of the underlying code are overdue for refactoring.
- A regression suite for `tktest` needs to be written.
- Change the sqlLite specific queries to use tdbc.
- The *Expected* value needs to support regular expression as well as glob wildcards.

7 Summary

The TkReplay application from 1995 was renamed `tktest` in 2004 when it was updated and expanded to support introspection and simple testing. This application was again expanded in 2015 to support automated regression testing for large, complex GUIs.

The `tktest` application suite is available on the Noumena Corporation website (`www.noucorp.com`) under the Tcl tab.