Tcl/Tk guidelines for improved automated regressions - a case-study

Saurabh Khaitan, Madhur Bhatia, Tushar Gupta
Mentor Graphics Corporation

Abstract:
Automating regression flow for any GUI application always poses a lot of challenges. The automation demands recording and replaying the user interactions with the GUI often with timing synchronization between operations. There are some commercial and open-source solutions available which provide interactive test capabilities which can capture user sequences. Such testing applications access the internal objects of the application through the hooks provided via GUI toolkit. For automating the GUI based tests for our emulation product Veloce, an enterprise application based on Tcl/TK, various testing solutions were evaluated. Automation of GUI testing was supplemented by other methods like tcl script regressions, custom test applications and specific logging wherever required. The paper presents a case-study of various issues encountered in developing good automated regression flow for emulation GUI, and presents some coding and development guidelines for Tcl/Tk applications in general for suitability for development of automated regressions.

GLOSSARY:
AUT- Application under Test
GUI – Graphical User Interface
VP – Verification Points are used by automated regression tools to check the states of GUI.
Waveform Widget – Widget to view waveform signals
Path browser – Widget to traces signal connections in a design.

1. Introduction:

With the increase in the size and complexity of applications over time the need to ensure functional compliance over releases is increasingly felt. This can be guaranteed by creating extensive and stable automated test suites. The difficulty in creating such regression suites multiplies manifolds when GUI testing is involved.

Even a small size application creates excessive permutations and combinations of sequence of steps that are too large to test manually in consistent manner. Specially when the applications are continuously evolving, keeping in pace to test each feature is a cumbersome task. Even elaborated manual testing processes are error prone and there are chances of test scenarios being left out. Automated GUI testing tools have the capability to automate this task for you and it helps you to improve the quality of your application. It also cuts down the "time to market" for the tool if deployed in early stages.

Earlier GUI automation tools were analog, which means they recorded the mouse movements using co-ordinate movements. This was a very poor automation technique and required tremendous maintenance efforts every time there was a minor change in the AUT. Modern GUI automation tools are smarter and are "object-based". These tools are closely integrated with the AUT and recognize the control in graphical applications like buttons, menus and text input widgets. This technique is more robust and requires no changes in the regression suites with changes in GUI design or screen resolution so we will use "object-based" tools.

While developing automated regression suites we encountered many issues that were resolved in the process. We discovered that if the planning and strategies are done in the early stages most of these challenges can be met easily. In this paper we discuss the guidelines that are followed to develop robust and reliable regression suites. These guidelines are captured in the following sections:

- Guidelines for developers (Section 2)
- Testing strategies for QA engineers.(Section 3)
- Limitations and their workaround (Section 4)

The guidelines suggested for developers are to be applied at the initial stages of development and are discussed in detail in the following sections. The testing strategies for QA engineers are to be used at the time of automated regression suite development. At the end the paper summarizes some limitations that we encountered even after following these guidelines. The paper also discusses the workarounds used to plug these limitations.

2. Guidelines for Developers:

To facilitate the development of robust and reliable regressions, the developers need to provide hooks to various GUI objects. Apart from this, the developer also may be required to create infrastructure to dump extra information for regressions. We list below some of the techniques we use to help building regressions for our tool.

2.1. Use of global arrays: Global arrays can be used to store widget names so as to provide access to internal GUI components. There can be multiple arrays and each of them may have handles to widget instances related to specific feature or module.

Another way to provide hooks is by providing access through functions for all the relevant widget instances.

Providing access to various widgets will help the test writer emulate the exact user behaviour. For example, if there is a button that calls a function, test writer can invoke the button rather than call the function directly if he has the access to the button.

2.2. Direct use of internal functions: We also found that direct invocation of functions in tcl script to perform a GUI operation to be useful in verification of the code. A sequence of such functions use, can emulate a complex user operation that is not easy to capture in a GUI regression tool. Therefore, while developing an application, developer should design his code in a way that code is encapsulated into small functions that can be used in the regressions to emulate complex user sequences.

The script shown below illustrates use of sequence of functions along with use of arrays, to perform a complex user action of creating an annotation file, adding signals to it, and saving the file.

```
project open -prj_file_path veloce.prj
createAnnotationFileWin

addNetToAnnotationWin top.AIn[1:0]
addNetToAnnotationWin top.BIn
addNetToAnnotationWin top.w1[9:6]
addNetToAnnotationWin top.YOut
addNetToAnnotationWin top.ZOut

$annotationWin(fileTypeCB) invoke
$annotationWin(fileTypeCB) selection set
1
$annotationWin(fileTypeCB) invoke

saveAnnotationFile
```

In the code above a complex user action is getting mimicked by calls to functions `createAnnotationFileWin` to create an annotation window, `addNetToAnnotationWin` to add multiple nets to the opened window, and finally saving the file using the function `saveAnnotationFile`. All these functions are internal functions and the GUI users are not aware of them.

2.3. Use of environment variables: For regressions, the use of environment variables can help to eliminate user interactive controls and provide behaviour suitable for regressions. This behaviour can be actions like printing the message to log files which can be used for automated verification. The coder should be judicious in the use of environment variables. If the checks are placed in the part of the code which is excessively used it can cause a performance penalty. Let's look at a sample code to understand the use of environment variables.

```
proc medmessageBox {args}
{
    …
    if (![info exists
::env(MED_REGRESSIONS)]}
    {
        if {![batch_mode]}
        {
            tk_messageBox
                -parent $par
                -message $msg
                -type ok
```

```
                -title $ttl
                -icon $icon
        }
        else
        {
            med_message $msg
        }
    }
    else
    {
        med_message $msg
    }
    …
}
```

In the code above we can see that with the use of environment variables like MED_REGRESSION the part of the code which will be active in regression mode is selected.

2.4. Use of algorithmic functions: The algorithmic functions in the code can be written in the manner that they can be used independently through the script to perform the tasks and test them. This will enable efficient test creation to test these functions in specific. With the use of environment variables they can be used to dump extra information for regressions.

For instance in our application we have a function `findpath` that is used internally to search a hierarchical path in a tree and update the tree to point to the path if it is found. Since this function can work as a standalone function, it is used in automated regressions. With help of environment variables it is able to print the status for regressions.

```
proc findpath {fullpath separator tree}
{
    # sanity checks
    …
    #tree search algorithm
    …
    # tree update
    …

    # code for regressions
    if {[info exists
::env(MED_REGRESSIONS)]}
    {
        if {$found}
        {
            set msg "path found"
        }
        else
        {
            set msg "path not found"
        }
        echo $msg
    }
}
```

In the code above the standalone function findpath is powerful enough to perform complex GUI operations like tree search, tree update. In regression mode this will dump extra debug information for verification.

2.4. Text representation of graphical display: Many times screen display, may have a text representation. For example waveform display can be efficiently represented as text file. Application developer can make sure that a way is provided to dump this type of display in a file which can be used in script based automated regression for verification. In our GUI we also have ways to dump display in schematic and path browser to a file. We also dump trees into text file for verification.

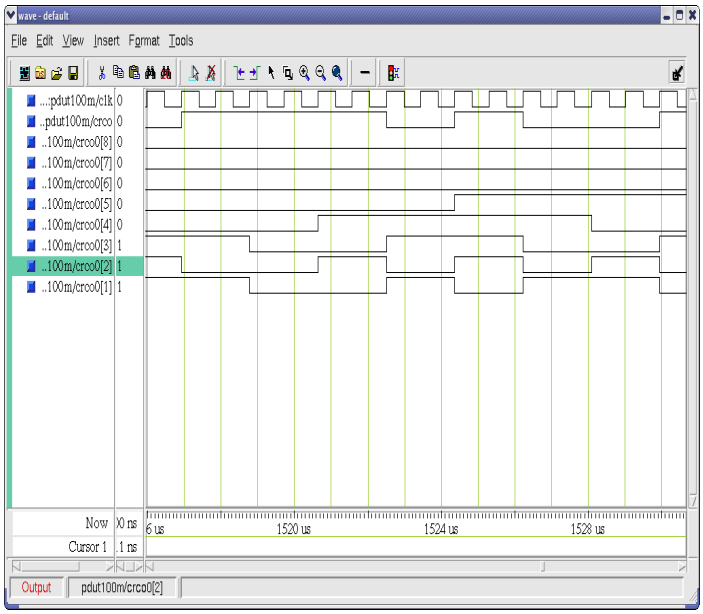

Figure 1: Waveform Widget

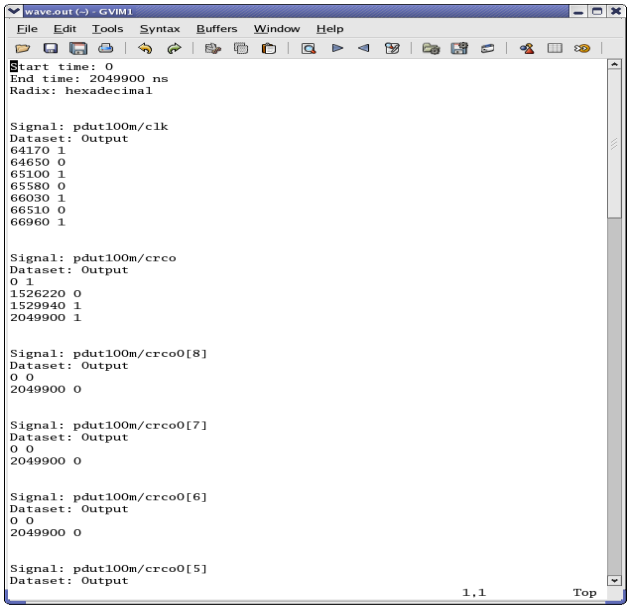

Figure 2 : Text representation of waveforms

The Figure 1 shows a graphical representation of waveforms as it is visible to the user. For regression purposes we are using a text dump of the waveform display as shown in Figure 2.

3. Testing strategies for QA Engineer

While creation of automated tests we developed some strategies which were very useful in making the regression suites resilient. To make the regression suites robust it is recommended that "Object based" GUI automation tool be used. Some guidelines that we followed are:

3.1. Use of Verification points: Just recording the user inputs to the GUI and re-playing them will not cover the testing requirements of GUI applications. It is of utmost importance that the state and the different properties of the numerous controls and widgets in the AUT are also checked. For incorporating these checks we used the concept of verification points. Verification points are checks introduced in the tcl script recorded through the GUI testing tool. These checks verify the current state and properties of the controls and widgets of the AUT after desired sequence of events. These checks are very important in ensuring the functional compliance of the AUT.

To illustrate this further lets take a scenario where a user is interested in checking the state of a widget. This can be easily done by querying the value of properties of the widget. In order to test the enabled state of a widget ($widget) the code will look like:

```
test compare [property get $widget state]
"enabled"
```

When we use Verification points these compare calls are inserted into the tcl script recorded by the automated GUI testing tool. Sample Code for checking state and image of a new_file button looks like:

```
# Verification Point 'newfile'

test compare [property get
[findObject
":vsim.dockbar.tbf0.standard.tb.button_0"
] state] "normal"

test compare [property get [findObject
":vsim.dockbar.tbf0.standard.tb.button_0"
] image] "__new_icon"
```

Here the automated testing tool inserted two property get calls, which then compare the returned value with the expected value. This technique is very useful in checking the functionality and state of the GUI.

3.2. Use of Global procedures: Creation of procedures for standard sequences in GUI has multi-fold advantage. Primarily it helps QA engineers to cut down on the time required for creation of tests as the same code can be leveraged across test suites and reduce maintenance overhead. It also helps in stabilizing the regression suites across incremental software releases. If there are any major changes in the application controls in the GUI code all the tests that are accessing that control of the AUT will start failing. With use of global procedures this can be achieved by changing at one place. The verification points which are part of the global procedures can also be shared across tests making testing of the states of the widgets in the AUT more exhaustive. Sample script using global proc:

```
proc main {}
{
        snooze 10

        #sourcing all global scripts
        source [findFile scripts
        "clean.tcl"]
        source [findFile scripts
        "analyze.tcl"]
        source [findFile scripts
        "close.tcl"]


        # global procedure - clean_all
        clean_all

        # global procedure - analyze
        analyze

        #global procedure - close_proj
        close_proj
}
```

In the sample script above it can be seen that we are calling three global procedures – clean_all, analyze and close_proj .We can see that usage of global procedures makes the test look very concise and manageable.

3.3. Synchronization points: Object based GUI testing tool insert time synchronization (snooze commands) to imitate user actions. This approach can make the regression suites unreliable at times. Consider a case when the test was recorded on a fast machine and is re-played on a slower machine, in such cases the test may fail. To avoid such scenarios we used the concept of synchronization points. While recording test the user can select to insert waitForObject statements. These statements wait for the given object to exist and be accessible thus eliminating the race condition created on slower machines. Sample Code:

```
waitForObjectItem ":vsim.#mBar" "File"
invoke activateItem ":vsim.#mBar" "File"

waitForObjectItem
":vsim.#mBar.#mBar#file" "New"

invoke activateItem
":vsim.#mBar.#mBar#file" "New"

waitForObjectItem
":vsim.#mBar.#mBar#file.#mBar#file#new"
"Project..."

invoke activateItem
":vsim.#mBar.#mBar#file.#mBar#file#new"
"Project..."
```

In the sample code above three synchronization points (`WaitForObjectItem` calls) were inserted to wait for the existence of the item before invoking clicks on them. The synchronization points can be used effectively in this manner to synchronize the automated tests.

3.4. Offline debug: It is very important for a GUI automation tool to provide its users with a way to debug the failures in an offline mode. This helps the QA engineers to debug the failures in the regressions. Modern GUI automation tools provide environment variables which when enabled will capture the state of the GUI when a failures occurs. So whenever a Verification point fails or GUI behaves unexpectedly the current state of the GUI is captured as screenshot by the automation tool. This screenshot can be viewed later in offline mode after the regression is over. This is a very useful for debugging of regression failures.

4. Limitations & their Workarounds

While the guidelines mentioned above enables the user to automate the testing of majority of the AUT, but still there are some limitations that exist with using the object based GUI testing tools. We have captured the limitations encountered and suggest some workaround to overcome them:

4.1. Custom widgets: Automated testing solutions are equipped to recognize and support standard Tcl/ Tk widgets. If the application uses custom widgets then the automated testing tool will fail to recognize the new widgets. There are two ways to solve this problem. The first and the robust solution is to work with the testing solution development team and get the support for the custom widget integrated in the GUI automation tool. This will at times require the user to share his custom widget code with the automation tool development team. The second solution is to use the conventional analog-style testing. This will enable the user to even create tests for custom widgets. Of the two solutions proposed the former solution is more robust and reliable.

4.2. Drag and drop: The testing solution that we employed for our test creation was not capable of capturing Drag and drop operations of the user. There are many black box GUI testing tools that are VNC based and capable of capturing DnD operations.

4.3. Canvas widget: Canvas widgets used in the AUT are similar to drawing canvasses. These widgets do not have properties like other standard widgets such as buttons. Thus it is not possible to create Property Verification points for canvas widgets as was done successfully for other widgets. To verify these widgets we used screenshot verification points. In screenshot verification screenshots of the canvas widget is captured at the time of test creation which is used as a reference image for consecutive runs. An example of the screenshot captured can be seen in Figure 3.

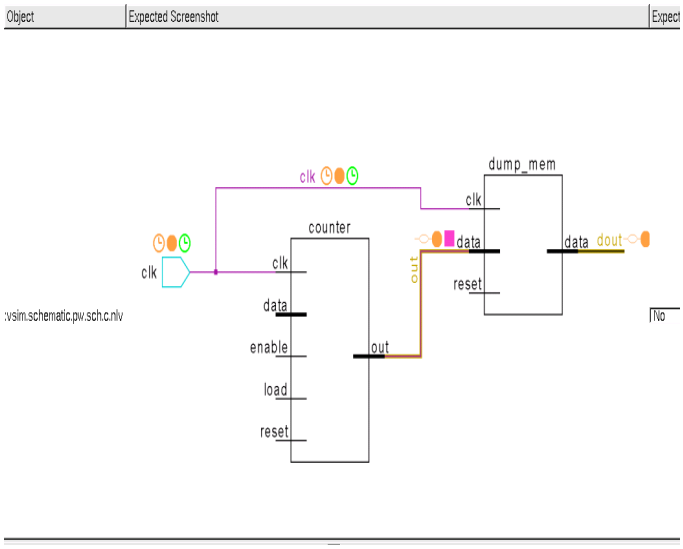

Figure 3: Screenshot Verification point

There is also an option of adding mask to remove unwanted portion of screenshot from these image comparisons. Other way of verifying canvas widgets is using textual dump of graphical display with the help of developers as discussed in detail in Section 2.4.

5. Conclusion:

Following the strategies elaborated above we were able to create robust regression suites for our Tcl/Tk based application. Initially some changes were required in the GUI code to align it to guidelines mentioned before but once the test suites were created it helped in eliminating the arduous process of manually testing the GUI. This not only enabled us to reduce the testing time for our GUI applications but it also caught issues which sometimes creep into subsequent patches of the software. The GUI automation testing tool that we used was Squish from Froglogic. When the tool is in the development phase if basic guidelines mentioned in the paper are followed majority of the GUI testing can be automated. Following the strategies discussed in the paper while test creation stable and robust automated regression can be developed.

6. Bibliography:

1.http://en.wikipedia.org/wiki/GUI_software_testing
2.Squish, http://www.froglogic.com/download/book4