Applying tcltest to Tk Applications

Robert W. Techentin, Sharon K. Zahn, Barry K. Gilbert, Erik S. Daniel
Mayo Foundation, Rochester, Minnesota, 55905, USA

Abstract— **This paper presents an approach for automating testing of graphical Tcl/Tk applications using the tcltest package. Some Graphical User Interface (GUI) test automation tools rely on screen coordinates and screen captures to drive the program under test, while others may drive at the widget level. This new approach tests the application at the procedural level, but supports starting and stopping the Tk windowing application multiple times in the same test file.**

I. INTRODUCTION

Tcltest, the testing package distributed with the Tcl programming language [1] [2] [3] [4] is an established and robust test harness supporting automated regression testing. Its inherent portability, flexibility and scriptability make it suitable for many platforms and environments. While tcltest was originally developed specifically for testing Tcl interpreters, it can be used to test procedures, objects, libraries and entire programs written in Tcl or accessible from Tcl, and its speed makes it possible to execute thousands of individual tests per minute.

To achieve high test throughput, tcltest runs all tests within a single test file in the same Tcl interpreter. While this approach provides excellent performance, it has the side effect of sharing the interpreter state between tests, and it is up to the test author to ensure that tests run independently. For application testing, it is often desirable to discard all state and restart the application. GUI applications based on the Tk toolkit are not well suited to restarting in the same interpreter.

This paper presents a new technique for writing tcltest test suites which exercise Tcl/Tk applications. Test scripts start the application in a slave interpreter, which loads Tk and runs the application. The test script has complete access to the internal application programming interface (API) and application state through the slave interpreter evaluation mechanism. When the test is completed, the slave interpreter is destroyed, destroying that instance of Tk. Each test script in the test file gets a ”clean” environment in a new slave interpreter.

In the following sections, this paper will review GUI application test approaches and discuss limitations of the tcltest package which make it difficult to test Tk-based programs. This paper's new technique, called the slave interpreter approach, will be presented and discussed with examples. After discussing the limitations of the slave interpreter approach, an application style will be suggested which could mitigate some of the shortcomings.

II. TCLTEST AND APPLICATION TESTING

The tcltest framework supports running suites of multiple test files, each of which usually contains many individual test scripts. Tcltest options and constraints control which test files and individual tests are run or skipped with a great deal of granularity, but most test files will run multiple tests in the same Tcl interpreter. Individual test scripts are coded to run independently, so that a skipped or failed test does not affect the rest of the tests in a file.

Tcltest has been employed extensively as a unit test framework, primarily for libraries and applications coded in or for the Tcl programming language. The preeminent test suite based on tcltest is the collection of over 19,000 unit tests that are distributed with the Tcl programming language and the Tk toolkit. Most of these test cases, like those supported by the various xUnit frameworks recently touted in the popular literature [5] are unit tests, validating the functionality of functions, classes or modules.

Tcltest can, however, easily support higher level integration testing or even system level (application) testing. The power of the Tcl packaging mechanism makes it possible for test scripts to include significant functional blocks, and exercise the interrelationships between them. Applications that have integrated a Tcl interpreter for script automation can take direct advantage of tcltest to exercise all functions that are available to the interpreter.

But tcltest has not been employed extensively to perform application-level testing. The ”usual” tcltest approach is to load script code or dynamically linked libraries into the same interpreter with the test suite. For application testing, however, it is often desirable to restart the entire application for each test. The DejaGNU test framework [6], for testing the GNU compilers, debuggers, and utilities, uses Tcl and Expect in this fashion. The applications are controlled through standard input, essentially emulating typing commands on a terminal.

GUI-based applications are, in general, not suited to control from a terminal, and often require special test harnesses which drive the GUI from either the window manager or GUI toolkit level. Test harnesses based on GUI techniques will usually start an instance of the application, and then inject a series of recorded keyboard and mouse events to drive the application to a known state. Test success or failure is determined by evaluating the screen or widget states or application state. A new test can restart the application and drive it to a different state. But Tcl and Tk are not well suited to re-initialization within a single interpreter. Including multiple, independent application-level test scripts in one test file is not straight forward, and without an effective means to restart the application, test files would be limited to a single test case each. In the case of Tk GUI application, it would be necessary to use a specialized tool for recording and replaying GUI interactions, even if the primary goal is exercising the application logic, and not the GUI.

III. GUI-BASED TESTING TOOLS

There are many GUI-based test tools available. Android [7], an open source tool using the X window event mechanism, and Winrunner [8], a commercial tool which runs only on Microsoft Windows, both rely on screen snapshots to determine test success or failure, which can be fragile. A change in screen resolution or font or window decorations can cause an entire test suite to fail, even when the application is operating correctly.

TkReplay [9] is tailored for testing Tcl/Tk applications, driving the application at the widget level, and interacting with the internal state of the application to evaluate test results. This makes it possible to code ”white box” tests in a framework like TkTest [10], evaluating the application's internal state, instead of relying on screen snapshots to determine success or failure.

But these GUI-driven techniques have a distinct disadvantage in that the test scripts usually have little in common with the code they are testing. A recorded list of keystroke and mouse events does not describe the test author's intent. In fact, most GUI tests must be recorded with a software tool, perhaps with documentation added later. Tcltest test scripts, on the other hand, are written by a tester and succinctly show the intent of the test and the expected results.

IV. SLAVE INTERPRETER APPROACH

To overcome tcltest's inadequacies, we have developed an application-level testing approach that creates a new slave interpreter for each test in the test suite. Each test is run in a new Tcl interpreter, and can load code and extensions (including Tk), define variables, perform I/O, and otherwise act independently of other interpreters in the same process. Each test can load Tk, build a GUI, and perform application operations under control of the test script, and exit, without affecting the master interpreter or the other tests.

This approach has several significant advantages. Testers can write ”system level” scripts which start and stop an entire Tk application without resorting to additional GUI driving tools. These scripts can exercise the application's internal API, which is not always feasible in unit testing.

As an example, consider ”Hello World!” program in Listing 1, coded in Tcl/Tk. Note that the functional logic of the program is contained in the hello procedure. This procedure is essentially the application's programmable interface (API), and if it were stored in a separate file, it would be simple to create unit tests with tcltest. But in this case (and many cases in ”real world” applications), the logic is embedded in the GUI, and testing the API requires running the entire application.

We could write a tcltest to exercise the hello procedure using the slave interpreter technique. The test would have to initialize the application, exercise the hello procedure, and finally destroy the application by deleting the slave interpreter, as shown in Listing 2.

The test setup code creates the slave interpreter and starts the application using the `source` and `update` commands, which emulates launching the program and waiting for the GUI to appear. The cleanup code simply deletes the slave interpreter, which destroys Tk, closes all the GUI windows, and also deletes all the program logic. The setup and cleanup logic could be common to many tests, and could be better implemented as a procedure in the test file, as shown in Listing 3.

```
proc hello {} {
    puts "Hello World!"
}

package require Tk
grid [button .b -text "Hello" -command hello]
```

Listing 1. Hello World example in file hello.tcl

```
package require tcltest 2

tcltest::test hello-1 {check return value} -setup {
    set app [interp create]
    $app eval {source hello.tcl}
    $app eval {update}
}  -body {
    $app eval {hello}
} -result {} -cleanup {
    interp delete $app
}
```

Listing 2. Hello World test

V. SLAVE INTERPRETER ISSUES

The slave interpreter execution environment is slightly different from that of the main interpreter. Several issues must be handled by the test setup and cleanup processes.

To emulate executing an application, the slave interpreter must `source` the application's main program file, but the global variables argv0, argv, and argc are not automatically defined. The test setup code should ensure that these values are defined in the slave interpreter before running the application.

The global environment array, env, is shared among all master and slave interpreters. This feature is useful for communi-

```
package require tcltest 2

proc startApp {} {
    global app
    set app [interp create]
    $app eval {source hello.tcl}
    $app eval {update}
}

proc stopApp {} {
    global app
    interp delete $app
}

tcltest::test hello-1 {check return value} -body {
    $app eval {hello}
} -setup {startApp} -cleanup {stopApp} -result {}
```

Listing 3. Test Procedures for starting and stopping application

cation between interpreters. But in the case of running multiple regression tests, we need to save and restore the environment before running each test. This is easily accomplished using Tcl's array commands in the test setup and cleanup procedures.

Each slave interpreter inherits the standard channels, stdin, stdout, and stderr, from the master interpreter. Output from the slave is directed to the same channels as the master, so the application will operate normally. Tcltest, however, intercepts both stdout and stderr by redefining the `puts` command, and records their contents for comparison when the -output or -errorOutput options are employed. The slave interpreter must have a similar replacement for the `puts` command so that the output comparisons operate correctly. A simple alias to the master interpreter's `puts` will not suffice, as this causes output to the application's other channels to fail.

Finally, the slave interpreter should not call `exit`, as this would exit all interpreters and terminate the test abruptly. An alias can be defined for the slave's exit command so that it correctly deletes the slave interpreter and performs other necessary cleanup. This exit procedure can also handle the optional return code, so that application tests can check for exit status.

The startApp and stopApp procedures can be generalized to accept command line arguments, and perform all the required initialization and cleanup for executing Tk applications. For the general case, they must also be augmented by exitApp and putsApp to handle application exits and standard channel output.

VI. LIMITATIONS OF THE SLAVE INTERPRETER APPROACH

The slave interpreter concept was originally applied to a personal project (the soccer coach's assistant), but the intent was to develop tests for complex electronics modeling applications being developed in support of advanced research programs. The generalized slave interpreter application routines were developed specifically to enable application testing of xm3: the third generation of a transmission line modeling system being developed at Mayo. Xm3 consists of about 7000 lines of Tcl, supported by 1000 lines of custom C code and a score of Tcl packages. Implementing slave interpreter tests revealed several shortcomings in the approach.

Creating a slave interpreter and sourcing the main program file for every test has about twenty times the overhead of a traditional tcltest procedure. Adding on the overhead of initializing Tk and drawing a window for every test is slower by another order of magnitude. Running the tests for an entire Tk application including building and destroying the GUI, could be as much as 1000x slower than traditional unit tests. It might be tempting to perform many different functions within a single test script, but that would reduce the granularity of the tests, and make errors more difficult to debug.

Fortunately, the slave interpreter approach could also be used outside of the individual test script setup and cleanup procedures, and groups of related tests could be run within a single instance of the application, as shown in Listing 5. These tests are for a soccer coach's team management application,

```
proc startApp {argv0 args} {

    # save environment
    global env envSave
    set envSave [array get env]

    #  create slave interpreter
    global app
    set app [interp create]

    # prevent exits
    $app alias exit exitApp

    #  redirect puts stdout/stderr to master
    $app eval [list namespace eval tcltest::Replace {}]
    $app eval [list rename puts tcltest::Replace::puts]
    $app alias puts putsApp

    # set up command line args and run app
    $app eval "set argv0 $argv0"
    $app eval "set argc [llength $args]"
    $app eval "set argv [list $args]"
    $app eval "source $argv0"
    $app eval "update"
}

proc stopApp {} {

    #  delete slave interpreter
    global app
    catch {interp delete $app}

    # restore environment
    global env envSave
    array unset env *
    array set env $envSave
}

proc exitApp {{returnCode 0}} {
    stopApp
    return $returnCode
}

proc putsApp {args} {
    global app
    switch [llength $args] {
        1 {
            # only string to be printed
            puts [lindex $args 0]
        }
        2 {
            # either -nonewline or channelID
            switch [lindex $args 0] {
                "-nonewline" -
                "stdout" -
                "stderr" {
                    eval puts $args
                }
                default {
                    $app eval tcltest::Replace::puts $args
                }
            }
        }
        3 {
            # both -nonewline and channelID
            switch [lindex $args 1] {
                "stdout" -
                "stderr" {
                    eval puts $args
                }
                default {
                    $app eval tcltest::Replace::puts $args
                }
            }
        }
    }
}
```

Listing 4. Generalized procedures for testing Tk applications

```
startApp soccerManager.tcl

tcltest::test player-1.1 {create a new player} {
    ...
}
tcltest::test player-1.2 {update a player phone} {
    ...
}
tcltest::test player-1.3 {delete a player} {
    ...
}

stopApp
```

Listing 5. Multiple tcltests for one application invocation

```
proc addPlayer {args} {
    set p [Player %AUTO%]
    if { [llength $args] > 0 } {
       $p configurelist $args
    } else {
        playerDialog $p
    }
    if { [$p valid] } {
        $::team add $p
    }
    return $p
}
```

Listing 6. Example of testable Application API procedure

written in Tcl/Tk. There are traditional tcltest unit tests for application objects and procedures, and the slave interpreter approach was used to create application level tests.

Some GUI implementations resist test automation using this approach. Dialogs constructed using the BWidget library, for example, use tkwait to suspend execution until the user closes the dialog. This is convenient for the application programmer, simply calling `Dialog draw`, which returns only after the user clicks on one of the dialog buttons. But these dialogs cannot be exercised by automated test scripts.

There may also be problems with some extensions that are not suitable for loading into multiple interpreters. Unlike pure Tcl extensions, compiled extensions are loaded into the process once using dlopen(). Just as some Tcl extensions are not thread safe, some compiled extensions might not initialize correctly for multiple slave interpreters. BLT 2.4z, for example, was released with a bug that prevents it from being used from multiple interpreters. The problem was corrected, and a patch can be downloaded from http://sourceforge.net/projects/blt/. But the pre-compiled Windows distribution predates the patch. Thus, the slave interpreter technique will not work on Windows with applications using the pre-compiled BLT library.

VII. PROGRAMMING FOR TESTABILITY

Proponents of agile development methods [11] suggest that tests should be developed before or during coding. Following this philosophy makes it reasonable to design the application in a style specifically for testability [12]. The slave interpreter approach is particularly well suited to architectures with an API as close to the GUI as possible.

It is theoretically possible to drive the GUI layer of the application using tcltest and the slave interpreter approach. If widget names are known, events can be synthesized directly in the tcltest script using `event generate`, and the internal state of the application can be inspected to deduce success or failure. But this level of testing is better handled by a tool like TkReplay, which specializes in capturing and replaying widget events.

Just below the GUI layer, at the Application API, are the procedures called directly by the GUI widgets (e.g., menu buttons). These procedures could be exercised by test scripts if they were designed to work around the limitations of the slave interpreter approach, and if they were made to be testable.

For example, in the soccer coach's application, one menu button calls the `addPlayer` procedure, which creates a new player object and adds it to the list of players. Since this application was implemented using BWidgets' Dialogs, this procedure must have the option of not actually drawing the dialog. Testability can be implemented using optional arguments for input, and a return value that can be validated. If the addPlayer function in Listing 6 is called with parameters, they are used to define the new player object. Otherwise, the dialog is drawn and the user (or a GUI test program) supplies the data values.

VIII. CONCLUSION

This paper presented a slave interpreter technique for testing Tk applications using tcltest. This technique fills a niche between unit testing, at which tcltest excels, and GUI testing using specialized test harnesses like Android or TkReplay. This new technique makes it possible to execute Tk programs from within test scripts, controlling them through the application API, and validating them by inspecting the application state.

While the procedures presented here were implemented in test script files, it would be relatively easy to implement them as a package which could be loaded in addition to the tcltest functionality. If the technique proves both useful and successful in the long term, it might make sense to extend the tcltest package to incorporate the concept of running application code in slave interpreters. Tcltest 2.2, presently shipping with the Tcl core 8.4, already supports options for running entire test suites (many test files) in the same process or in individual processes. This concept of running tests in individual interpreters would be a natural extension to that concept.

ACKNOWLEDGMENT

The authors would like to thank E.J. Hanlon for technical assistance, and T.J. Funk, K.C. Garry, and D.D. Windhorst for manuscript preparation.

REFERENCES

[1] Tcltest and Tcl/Tk are available at http://www.tcl.tk/

[2] John Ousterhout, ”Tcl and the Tk Tookit,” Addison-Wesley Professional, March 1994.

[3] Brent Welch, Ken Jones, Jeffrey Hobbs, ”Practical Programming in Tcl and Tk (4th Edition),” Prentice Hall, June 2003.

[4] David N. Welton, ”Make it right using Tcl: Software Testing with Tcl for Apache Rivet,” Free Software Magazine, May, 2005.

[5] Panagiotis Louridas, ”JUnit: Unit Testing and Coding in Tandem,” IEEE Software, vol. 22, no. 4, July/August 2005, pp 12-15.

[6] Don Libes, ”Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs,” O’Reilly Media, November 1996.

[7] Larry Smith, Cameron Laird, ”Android: Open-Source Scripting for Testing and Automation,” Dr. Dobb’s Journal, July, 2001, http://www.wildopensource.com/activities/larry-projects/android.php

[8] Winrunner is available from Mercury Interactive, http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/

[9] Charles Crowley, ”TkReplay: Record and Replay in Tk”, Proceedings of the Third Tcl/Tk Workshop, July, 1995, http://www.cs.unm.edu/%7Ecrowley/papers/replay.tk95.html

[10] Clif Flynt, ”TkTest: Cross Platform and Remote GUI Regression Testing,” Proceedings of the 11th Tcl/Tk Conference, October 2004, http://www.tcl.tk/community/tcl2004/Papers/

[11] Kent Beck, ”Extreme Programming Explained: Embrace Change,” Addison-Wesley, October 1999.

[12] Antonia Bertolino, Paola Inverardi, Henry Muccini, Andrea Rosetti, ”An Approach to Integration Testing based on Architectural Descriptions”, Proceedings of the third IEEE International Conference on Engineering of Complex Computer Systems, September 1997, pp 77-84.