# A Regression Test Engine

An Architecture to Support
Automated Regression Test Suites

Chuck Matthews
Fifth Generation Systems, Ltd.

## Introduction

Most commercial hardware and software products are characterized by a product life-cycle that undergoes numerous evolutions and releases. Sometimes these release cycles result in completely independent products, but more often these cycles appear to be incremental advances in features and functionality for a product. A major factor in determining customer satisfaction through multiple releases of a product is the ability of the development organization to keep from breaking functionality that isn't supposed to change.

To manage this engineering task requires that the development organization have the proper tools to assist them in verification and validation of product functionality. For most organizations this means that an automated regression system is an absolute necessity for managing their business effectively.

In this paper I describe the architecture for a regression test system. There is nothing proprietary (or even terribly ingenious) about this architecture. I have used this basic design on multiple projects. The first implementation occurred in a multi-threaded C++ application that used the Microsoft MFC architecture. The next implementation was a generic C++/Tcl application that did not use the Microsoft MFC classes and was developed for cross-platform operation on Windows and various Unix operating systems. This proposal implements the design in the Centura 4GL database language. The design is equally extendible to numerous other languages as well.

## Regression Process

During the initial development phase, there may be some time period when the design is highly unstable and unsuitable for any regression testing activities. The development organization should minimize the amount of time that the regression tests cannot be run as much as possible. When the regression suites are activated, the normal process would be to run all or a subset of the regression suites on a daily basis. This process corresponds to the "daily build and test" runs that have been documented for a number of large development organizations.

A regression suite is a collection of individual test cases. Usually the test cases in any regression suite are related in some fashion, e.g. a common functional area that they test. Each test case will normally create a log file and one or more output dump files. The log file contains the sequential list of the test commands that executed and the result for each command. The dump file contains output data from the test case execution.

After the regression suite completes, the test manager runs a diff/compare program that compares the log and dump files from the current test run to a previous "golden" version of these files. When the current and golden versions of the files differ, then something in the implementation has changed since the golden version of the test case output was created. In this way the regression process flags all differences as "regression failures". A

tabulation program tabulates the regression results and lists all regression test failures. Someone from the engineering staff must check the regression results and investigate the failures the next day.

## *Implementation*

The test engine implementation has three major components.

- The *test command language* defines the commands that are necessary for exercising and validating the functionality of the design. Although the intent is to define appropriate commands for testing as much functionality as possible, it may not be possible to define a language that can completely test everything. Some design behavior may be difficult to test programmatically. Such behavior is usually not testable by automated means.
- The *test command process loop* reads the test case file, parses the file into the test command language, executes the test commands, and handles error conditions. After each test command completes, the next command is read, parsed, and executed. However, some test commands may need to wait until all windows event messages are processed before the next command can execute. For example, any test command that results in a new window being activated is not really finished until the new window has actually been created and finished processing the WM_PAINT message. Thus, for those commands the process loop must wait until the windows event loop is idle before processing the next test command.
- Detecting when the *windows event loop* is empty is easier in some languages than it is in others. Finding the correct method may require some ingenuity, but it is necessary in order to know when it is safe to continue processing the next test command.

## Test Command Process Loop

The test command process loop is a continuous programming loop that processes test commands until one of them needs to wait until the event loop is empty. In the Centura application, this loop is:

```
Set frmTest.bRC = frmTest.ProcessNextCmd()
While frmTest.bRC AND NOT frmTest.bSingleStep
  !
  ! Loop until you need to wait for a command to complete.
  !
  Set frmTest.bRC = frmTest.ProcessNextCmd()
```

The frmTest.ProcessNextCmd function reads, parses, and executes the test commands. A return value of TRUE allows the next test command to execute. A return value of FALSE requires the code to fall out of the loop and wait for the next time when the windows event loop is empty.

## Interaction with the Windows Event Loop

In an MFC application, the programmer can override the CWinApp::OnIdle function. This member function executes when the event loop is empty.

In the Tcl language, the programmer can create a timer handler:

```
Tcl_CreateTimerHandler(4, OnIdleAcsProc, (ClientData) 0);
```

The function OnIdleAcsProc will execute when the timer event expires. The function OnIdleAcsProc checks the event loop to determine whether it is empty. If not, another timer event activates. This process continues until the event loop is empty – which will eventually happen. When the event loop is empty, the next test command process loop can process the next test command.

In Centura, I chose to post a message to the application object when I want to wait for the event loop to become empty.

```
Set GnAppThreadID = GetCurrentThreadId()
Call PostThreadMessageA( GnAppThreadID, PAM_OnIdle, 0, 0 )
```

GetCurrentThreadId and PostThreadMessageA are Win32 API functions. At the top-level application object, the PAM_OnIdle event checks the windows event queue. If any posted messages exist, the programmer continues to post the PAM_OnIdle message to the application until the event loop no longer contains any other posted messages. At this time it is safe to process the next test command.

Most programming languages have some corresponding mechanism to determine when the event loop is empty. The test engine requires that the programmer find the appropriate check and implement it.

## Command Language

The test command language is defined to test the functionality of the application. Although some commands will be unique for an application, some test commands are common across numerous applications.

The following commands are the initial list for defining the test command language for the Gas Compliance application. I have not completed the definition of all test commands, but these commands are most of those that I expect to need.

**cm_dump_file <file_name>**
Open a file for subsequent dump commands.

**cm_dump <win_name>**
Dump the contents of the window win_name to the current dump file.

**cm_click <win_name> <ctrl_name>**
Simulate a mouse click over the control ctrl_name in window win_name, e.g. a pushbutton.

**cm_dbl_click <win_name> <ctrl_name>**
Simulate a mouse double click over the control ctrl_name in window win_name.

**cm_toolbar_click <win_name> <ctrl_name>**
Simulate a mouse click on the control ctrl_name in the toolbar of the window win_name.

**cm_select_item <win_name> <ctrl_name> <item_name>**
Select an item (item_name) in a list box or combo box. The control is ctrl_name in the window win_name.

**cm_toggle_expansion <win_name> <ctrl_name>**
Expand/Compress the currently selected item in the outline control ctrl_name of window win_name.

**cm_select_tab <win_name> <tab_ctrl_name> <tab_name>**
Select the tab tab_name in the tab control tab_ctrl_name in window win_name.

**cm_exit**
Close all files and call SalQuit to close the application.

**cm_single_step <TRUE | FALSE>**
Start/Stop single-stepping in the window frmTest

## *Appendix: Sample Test Case*

### Test Script File: test.cmd

```
//
//  Batch test script
//
cm_dump_file  testDump.txt
//
//  Dump the main window
//
cm_dump frmMain
//
```

```
//  Select Exception Processing
//
cm_single_step TRUE
cm_click frmMainMenu pbExceptionManager
//
//  Select the Four Rivers location
//
cm_select_item dlgSelectKeyedLocation lbKeyedLocation 'AMEREN CIPS'
cm_toggle_expansion dlgSelectKeyedLocation lbKeyedLocation
cm_select_item dlgSelectKeyedLocation lbKeyedLocation 'FOUR RIVERS'
cm_dump dlgSelectKeyedLocation
cm_click dlgSelectKeyedLocation pbOk
//
//  Refresh the exceptions window and select an item
//
cm_toolbar_click frmExceptions pbRefresh
cm_select_tbl_row frmExceptions.tblExceptionList 2
cm_dump frmExceptions
cm_invoke_tbl_row frmExceptions.tblExceptionList 2
//
//  The Work Detail tab should activate.
//
cm_dump frmExceptions
cm_select_tab frmExceptions picTabs Exceptions
cm_invoke_tbl_row frmExceptions.tblExceptionList 3

cm_dump frmExceptions
//
//  Exit
//
cm_toolbar_click frmExceptions pbCancel
cm_click frmMainMenu pbExit
cm_click frmMain pbAppExit
```

## Test Log File: test.log

```
Log file opened:

cm_dump_file  testDump.txt
cm_dump frmMain
cm_click frmMainMenu pbExceptionManager
cm_select_item dlgSelectKeyedLocation lbKeyedLocation 'AMEREN CIPS'
cm_toggle_expansion dlgSelectKeyedLocation lbKeyedLocation
cm_select_item dlgSelectKeyedLocation lbKeyedLocation 'FOUR RIVERS'
cm_dump dlgSelectKeyedLocation
cm_click dlgSelectKeyedLocation pbOk
cm_toolbar_click frmExceptions pbRefresh
cm_dump frmExceptions
cm_toolbar_click frmExceptions pbCancel
cm_click frmMainMenu pbExit
cm_click frmMain pbAppExit
```

## Test Dump File: testDump.txt

```
Dump file opened.

_____________________________________________
Dumping frmMain
Dumping list box: lbsStatus
  Initialize variables for ACCESS..
  Connecting to  database...
  Checking database version...
  Version 4.0.12 validated.
  Checking user limit...
  User limit not exceeded, 9920 remaining.
  Checking access level...
  Access level 2 Full...
  Compliance checks -- Optional...
  Loading Work Rules...
  ImagePath...   C:\GCS\IMAGES
  System Header -- Ameren CIPS
  System coordinates -- Optional...
  Minimum Allowed Date -- 01/01/1900 ...
s Initialization Complete!

_____________________________________________
Dumping dlgSelectKeyedLocation
Dumping list box: lbKeyedLocation
  AMEREN CIPS
  EAGLE VIEW
  EASTERN TECHNICAL SUPPORT
s FOUR RIVERS
  HERITAGE
  NORTHERN PRAIRIE
  SHAWNEE
  SOUTHERN TECHNICAL SUPPORT
  WABASH
  WESTERN TECHNICAL SUPPORT

_____________________________________________
Dumping frmExceptions
dfID:
dfnDue:
dfdDue_Date:
dfnComply:
dfdCompliance_Date:

tblExceptionList:
CTS, 28006072, TRANS/IC PROT MAIN, HANDHELD, TS-MISMATCH DETAIL,
09/11/2001, 12/11/2001, 9391170272, 180, 2773781238, 2000-09-11-
13.29.07.000000, C29700, 2000-09-14-8.05.39.000000
CTS, 28006071, TRANS/IC PROT MAIN, HANDHELD, TS-MISMATCH DETAIL,
09/11/2001, 12/11/2001, 7177802989, 180, 1774088521, 2000-09-11-
13.03.36.000000, C29700, 2000-09-14-8.05.23.000000
CTS, 28006069, TRANS/IC PROT MAIN, HANDHELD, TS-MISMATCH DETAIL,
09/11/2001, 12/11/2001, 2458826737, 180, 9540057516, 2000-09-11-
12.54.55.000000, C29700, 2000-09-14-8.04.53.000000
```

CTS, 26002121, TRANS/IC PROT MAIN, HANDHELD, TS-MISMATCH DETAIL, 09/11/2001, 12/11/2001, 3575835254, 180, 8611358345, 2000-09-11-12.16.30.000000, C29700, 2000-09-14-8.03.47.000000
CTS, 26002120, TRANS/IC PROT MAIN, HANDHELD, TS-MISMATCH DETAIL, 09/11/2001, 12/11/2001, 794643781, 180, 6827603610, 2000-09-11-11.04.19.000000, C29700, 2000-09-14-8.03.29.000000
, , , , , , , , , , , ,