MULTI-PLATFORM SCADA GUI REGRESSION TESTING AT CERN

P.C. Burkimsher, M. Gonzalez-Berges, S. Klikovits, CERN, Geneva, Switzerland

Abstract

The JCOP Framework is a toolkit used widely at CERN for the development of industrial control systems in several domains (i.e. experiments, accelerators and technical infrastructure). The software development started 10 years ago and there is now a large base of production systems running it. For the success of the project, it was essential to formalize and automate the quality assurance process. This paper will present the overall testing strategy and will describe in detail mechanisms used for GUI testing. The choice of a commercial tool (Squish) and the architectural features making it appropriate for our multi-platform environment will be described. Practical difficulties encountered when using the tool in the CERN context are discussed as well as how these were addressed. In the light of initial experience, the test code itself has been recently reworked in object-oriented style to facilitate future maintenance and extension. The current reporting process is described, as well as future plans for easy result-to-specification linking. The paper concludes with a description of our initial steps towards incorporation of full-blown Continuous Integration (CI) support.

INTRODUCTION

Many of the systems of the CERN Large Hadron Collider (LHC) accelerator [1] and the sophisticated physics experiments placed around its circumference are controlled using commercial Supervisory Control and Data Acquisition (SCADA) technology [2]. These production control systems have an expected lifetime upwards of 20 years, nevertheless are in general implemented by an ever changing stream of developers who frequently work on many other different sub-systems too. The CERN Joint Controls Project's JCOP Framework [3] was introduced into this mix to minimize the regular hurdles faced by new developers. The JCOP Framework strives to hide much of the complexity of the underlying SCADA tool used (WinCC-OA from Siemens, previously known as PVSS [4]). The Framework further aims to provide a level of consistency in the final application, in order to minimize the cost of long term maintenance as well as for operational safety reasons. The Framework comprises not only code segments (libraries) but also a substantial amount of Graphical User Interface (GUI) [5] software. This paper outlines the deliberate steps taken to ensure that the GUI implementations are rigorously tested.

QUALITY ASSURANCE

A strict quality assurance strategy is necessary not only to deal with the long project lifetime and staff rotation issues, but also with upgrades of the JCOP Framework itself and the packages upon which it depends (e.g. operating system version, SCADA tool, compilers, database clients and servers).

GUI TESTING

Early on in the Framework project it was recognized that the GUI mechanisms comprised a significant proportion of the software being generated. Programmers are used to testing library software in an automated fashion, but GUI regression testing is often not included [6]. To have full confidence in all of the modules of the Framework package, it is necessary to automate testing of their GUI aspects also. Considerable effort has therefore been invested in GUI regression testing – on a daily basis ensuring that the individual packages and components still behave the same way they did yesterday. This approach has proved to be very fruitful, especially when new developers first start maintaining code. Inadvertent side-effects of any changes made have been identified within 24 hours.

IMPLEMENTATION

In practice, we are testing WinCC-OA panels. An initial investigation using the tool "Rational Robot" [7] was dropped when ETM [8] switched the WINCC-OA windowing system to Qt [9]. A specialised product for the automated testing of Qt GUI interfaces was selected called Squish [10], marketed by Froglogic. Like Qt itself, Squish is available on many platforms, including those used in control systems at CERN, namely Windows and Linux. Out of the box, Squish allows you to record GUI input and test the properties (including existence) of screen objects.

Squish Recording Facility

Our first implementation made extensive use of the Squish recording facility. Squish generates human readable source code that, when executed, will replay mouse gestures and keyboard input. We opted to produce Python [11] code and have grown to appreciate the power and flexibility of this language. Although Python is interpreted, for this application domain there has never been a performance issue.

In practice though, our first implementation was very hard to maintain. Our surrounding environment was in a state of constant and uncomfortably rapid flux. WINCC-OA is upgraded to a new version each year or so. Although ETM strove very hard (and were successful) in maintaining functional backwards compatibility, their underlying implementation did change and this unfortunately was visible to Squish and the testing software. Underneath WINCC-OA, Qt itself has been undergoing a major version change also. As our test suites grew in size, more and more code needed to be changed to keep the test system working with each new version. Similar issues presented themselves with each new version of Squish. Although in general each new Squish version would bring highly desirable functional enhancements, it was usually necessary to modify and sometimes completely regenerate sections of Squish code. Our naive approach simply did not scale.

Code Refactoring

It became clear that we needed to refactor our code to make it more robust. We tackled this problem by extracting repeated sequences of recorded code into libraries, Fig. 1.

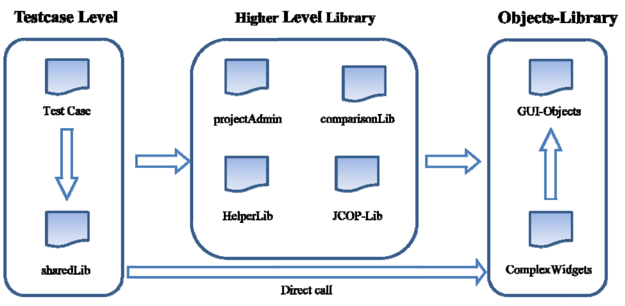


Figure 1: Library Structure.

The first step was to extract all of the routines which interact with GUI-widgets into their own library. The library was implemented in object-oriented style [12]. This programming paradigm makes it easy to define an instance of a GUI-widget once and to re-use it in several places. Our library contains several classes (one for each GUI-widget).

We then extracted all duplicate code from the test scripts into several higher-level application libraries dependent only on the above GUI-widget library. One example of an application level library would be the "Project Administration" library. This library contains a class with methods for creating, starting, stopping and deleting WINCC-OA projects, namely the functionality made available in ETM's "Project Administrator" panel. The code of these application library routines is used in nearly every test case. Extracting this code into a library makes test cases more robust and maintainable in case of changes of WINCC-OA.

The test case scripts themselves were thus reduced to library calls and verifications. Development of new tests is now quick and easy, as test creators can build on pre-existing functionality.

ENVIRONMENT

An important aspect of testing the Framework relates to the fact that the Framework runs on 2 major platforms, Windows and Linux.

One System Running On 2 Platforms

Our initial implementation attempted to validate the software on both platforms by producing a system that would execute on each of them. We opted to invoke our tests (with all of their file-system specific definitions) from within the Bash shell [13], running natively on Linux [14] but using Cygwin [15] on Windows, Fig. 2.

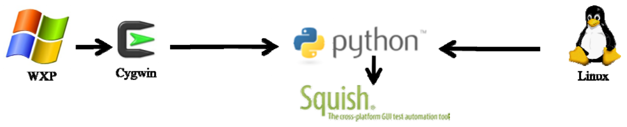


Figure 2: Platform commonality through Bash.

Although Cygwin is extremely powerful, the two Bashes are not identical and cause significant problems. Regularly updated versions of Cygwin provoked us into looking for an alternative solution.

One System Controlling A 2nd Platform

Squish itself is implemented as a client-server architecture, Fig. 3.

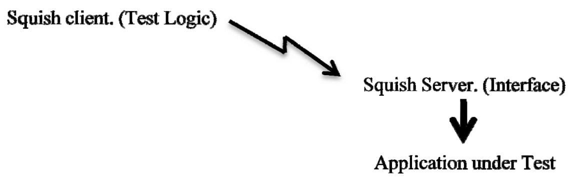


Figure 3: Squish is Client-Server.

A squish client executes the logic of the test scripts and a server executes the program being tested. This alternative arrangement was deemed to be appropriate for our multi-platform target scenario. We learned the hard way that this was not a particularly good idea. We decided that the client code would always run on Windows and would connect to the server on Windows or Linux as appropriate. Although this superficially meant that there was only one single body of testing code to maintain (always on Windows), in practice the code differed rather significantly depending on which was the target platform. As before, our problems derived from file system issues: Where was the program that was being tested going to get its data from and write its results to? These issues were not insurmountable, but they certainly made the code somewhat messy.

We encountered another much more practical problem which finally sounded the death knell to the "client-server on different machines" approach. The CERN Windows networked file-server environment was insufficiently stable to be able to work in this way. From time to time network connections from Windows to Linux would receive a very slow response, slow enough to cause the applications and particularly their display (through the Exceed tool [16]) to decide that the network had broken. All of the windows on the target machine(s) would collapse and all of the state of the system would be lost. Any test had to be restarted from scratch – and there was no guarantee that the new attempt would not suffer a similar fate. We even tried displaying in a virtual session and viewing with VNC [17] - but this was only of partial help.

Time For A Re-think

We therefore took advantage of the code rewrite opportunity described above to revert to an architecture where the Squish client (the test logic) and the Squish server (executing the code being tested) were once again executed on the same target machine. We abandoned Bash and rewrote the test harness completely in Python, which not only made it more compatible across platforms but also made it consistent with our use of Python for the Squish test scripts.

At the beginning of the project, we were using physical Windows and Linux machines as targets for testing. We are now migrating to using Virtual Machines, hosted in the CERN Computer Centre. The advantage for us is in regard to the provision of space to house, power and cool the machines. The advantage for CERN is that our virtual machines run their jobs once per day and then go idle, releasing VM server capacity for other useful work. Thus far, our experience of this service has been excellent, with our virtual machines being functionally indistinguishable from real hardware.

The move to virtual machines has neither improved nor made worse another problem we have commonly encountered: GUI testing can be very sensitive to timing issues. Running Emu in a different environment can introduce significantly different timing behaviour. Even though Squish implements a waitForObject() function which can deal with many of the synchronisation issues, we frequently see examples where the screen update is out of step with internal data. Clicking to open up a tree structure can appear to have completed, but pressing the next mouse click too quickly can still cause problems. It is tempting to sleep for a couple of seconds or so to give the system time to catch up – but how many seconds is enough? And how many seconds is enough on all platforms, so that the code doesn't suddenly break when you change to a new operating system (Windows XP to Linux or Windows 7) or to a new software release?

ORCHESTRATING THE TESTS

At the time of the code rewrite, we stepped back, took a wider view of the system and decided to incorporate a Continuous Integration (CI) tool [18][19] that would manage execution of our tests across multiple machines, operating systems and versions. We considered using Bamboo [20] from Atlassian as a CI tool as it integrates cleanly with the other Atlassian tools (Jira, Confluence) already in use at CERN. Bamboo however needs a license and initial investigations showed it to be rather rigid in its implementation. Instead, a more configurable (and free) Open Source equivalent was selected called Hudson [21].

Hudson offers us the possibility to centrally manage the testing whilst physically distributing it to multiple machines with different operating systems and software versions installed. Tests are triggered from a central (Windows) "Hudson Master Machine" onto one or more dissimilar remote platforms, Fig. 4. The Master machine runs a web server and Hudson is configured and controlled through any browser.

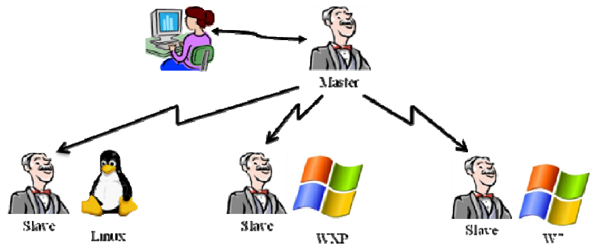
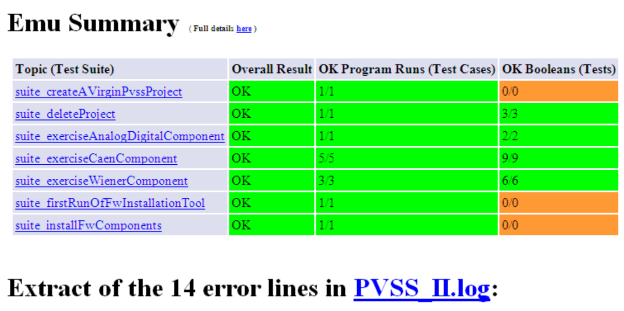


Figure 4: Hudson Master-Slave architecture.

An added bonus is that Hudson provides us with a convenient interface for having results returned to one central machine from where reports can be generated for sending to interested parties, Fig. 5.



Emu Summary (Full details here)

Topic (Test Suite)	Overall Result	OK Program Runs (Test Cases)	OK Booleans (Tests)
suite_createAVirginPvssProject	OK	1/1	0/0
suite_deleteProject	OK	1/1	3/3
suite_exerciseAnalogDigitalComponent	OK	1/1	2/2
suite_exerciseCaenComponent	OK	5/5	9/9
suite_exerciseWienerComponent	OK	3/3	6/6
suite_firstRunOfFwInstallationTool	OK	1/1	0/0
suite_installFwComponents	OK	1/1	0/0

Extract of the 14 error lines in PVSS_II.log:

Figure 5: Example result email with hyperlinks.

To date, the same central team that developed the Emu harness has also implemented the tests according to specifications drawn up with the Framework Component developers. This mode of working has several advantages:

1. The tests are prepared independently of the developers by a different individual, who is therefore unlikely to be aware of some of the implicit assumptions in the minds of those same developers. As such it is considered that these tests may be more likely to find errors than tests prepared by the developers themselves.
2. There is a written specification for the test scenarios – and consequently these are well documented. This may not always be the case in a situation where tests are written by the developers themselves.

Unfortunately this approach has a cost. The testing team becomes a bottleneck as all test code must first be written by them. For this reason we are planning to move to a scenario where the developers themselves will be able to plug their own tests into the Emu harness. Whilst alleviating the bottleneck, this comes at the cost of negating the advantages just described as well as having implications for the licensing of the software tools being used!

STATUS REVIEW

The theoretical advantages [18] of finding issues (bugs, incompatibilities, etc) before software is released, have once again been achieved in practice. New problems (side-effects) inadvertently introduced by new programmers joining the Framework development team have been spotted during the very next overnight run. We are able to test the JCOP Framework against new versions of WINCC-OA very readily by creating a new test target machine. This work is currently being extended to include the testing of a greater number of components from the JCOP Framework and also with new tests for the UNICOS Framework [22].

CONCLUSIONS

A scalable testing architecture has been defined and implemented. Automatic tests are executed every night on a variety of platforms. The architecture is not restricted to JCOP and is already being rolled out to encompass other frameworks at CERN.

AUTHOR CONTACTS

paul.burkimsher@cern.ch; manuel.gonzalez@cern.ch; stefan.klikovits@cern.ch

REFERENCES

Ask your favourite search engine for the most up-to-date links, or try the following:

[1] LHC http://public.web.cern.ch/public/en/lhc/lhc-en.html.
[2] SCADA http://en.wikipedia.org/wiki/SCADA.
[3] JCOP Framework http://j2eeps.cern.ch/wikis/display/EN/JCOP+Framework.
[4] WinCC OA (PVSS) http://www.etm.at/index_e.asp?id=2&amp;m0id=6.
[5] GUI http://en.wikipedia.org/wiki/Graphical_user_interface.
[6] An Integration testing Facility for the CERN Accelerator Controls System, N. Stapley, M. Arruat, J.C. Bau, S. Deghaye, C. Dehavay, W. Sliwinski, M. Sobczak, ICALEPCS 2009 https://espace.cern.ch/be-dep/CO/ICALEPCS%202009/1245%20%20An%20Integration%20Testing%20Facility%20for%20the%20CERN%20Accelerator%20Controls%20System/THP085-Paper-FINAL.pdf.
[7] Rational Robot http://www-01.ibm.com/software/awdtools/tester/robot/.
[8] ETM. http://www.etm.at/.
[9] Qt. http://qt.nokia.com/products/.
[10] Squish. http://www.froglogic.com/products/index.php.
[11] Python. http://www.python.org/.
[12] Wegner, 1990; Peter Wegner, Brown University [June 1990]: “Concepts and Paradigms of Object-Oriented Programming”, Expansion of Oct 4 OOPSLA-89 Keynote Talk.
[13] Bash http://en.wikipedia.org/wiki/Bourne-again_shell.
[14] Linux http://linux.web.cern.ch/linux/scientific5/.
[15] Cygwin “http://www.cygwin.com/”.
[16] Exceed http://connectivity.opentext.com/products/exceed.aspx.
[17] VNC http://www.realvnc.com/.
[18] Continuous Integration. http://en.wikipedia.org/wiki/Continuous_integration.
[19] Fowler, 2006; Martin Fowler [01 May 2006]: “Continuous Integration”, http://martinfowler.com/articles/continuousIntegration.html#BenefitsOfContinuousIntegration.
[20] Bamboo. http://www.atlassian.com/software/bamboo/.
[21] Hudson. http://en.wikipedia.org/wiki/Hudson_(software).
[22] UNICOS http://cern.ch/wikis/display/EN/UNICOS.