Automated testing of simulation and embedded software

Reinhard Knoll        Rory Yorke
Denel Dynamics, Nelmapius Road, Centurion, 0157
rory.yorke@deneldynamics.co.za
+27 12 671-1911

Copyright (C) 2015 by Denel Dynamics. Published and used by INCOSE SA with permission.

Abstract. In this paper we propose a system for automated testing of simulation and embedded software. Automated tests make thorough, frequent testing easy, which means errors will be caught more quickly. Automated testing also means that more of the engineer’s time can be spent on development, and that development can continue closer to field tests. In the proposed system, source code check-in triggers a chain of tests. These tests would include static checks, compilation, and test execution. For embedded software, we propose extending the automated testing to execution on digital hardware set aside for testing purposes, and ultimately to complete unattended Hardware in the Loop Simulation (HILS) execution.

Software testing

Software testing is a broad term, but in this paper we will concentrate on two types of testing that are automatable: test by execution, and static analysis.

The optimist tests software in order to check that it satisfies expectations (that is, specifications); in this view, “A bug is a test case you haven’t written yet” (Pilgrim 2004). Combinatorial explosion in input space leads the pessimist to Dijkstra’s view that “Program testing can be used to show the presence of bugs, but never to show their absence!” (Meyer 2008). This is true, but software testing remains a practical means to

- verify an implementation against specification, however superficially;
- allow implementation changes; if a test (or suite thereof) passes before and after optimizing an algorithm implementation, there is a good chance that the change is successful;
- and capture development knowledge, for instance if the test suite is expanded every time a bug is found.
Software testing does not mean that other assessment techniques, such as code review, are unnecessary. It is unfortunately easy to write code that passes tests and follows naming conventions, etc., but is still difficult to read and maintain. Test automation means reviewers don’t have to deal with mundane problems such as naming conventions, and can focus on how clearly implementation expresses intent.

Something separating many types of software testing from code review is the possibility of automation; that is, tests can be run, without human initiation, any time a software change is made. Automation relieves the developer of the responsibility of test execution, which means they can be run more often, and can be more extensive; automation also allows repeatability.

All of this comes at a cost, namely in developing the tests themselves, and developing and maintaining the automation infrastructure. These costs can be reduced by using existing test libraries (e.g., Google Test (Google)) and test managers (e.g., Jenkins (Kohsuke Kawaguchi)).

The cynic might note that this all amounts to testing one complicated set of software with another, equally complicated set of software, presumably doubling the total number of bugs. This criticism has merit; the benefits of this approach will not be reaped if such a project is initiated at the end of development; in this case the software to be tested already exists, will have been tested to some extent, and engineers will probably spend more time fixing the tests than the product.

However, if an automated test system is set up at the beginning of the project, once it is working engineers can focus on product implementation. In this case tests are developed with the product, and continue to be used throughout the development period. As field test deadlines approach, engineers don’t need to budget as much time for software testing, since tests already exist and are being executed all the time.

Complex systems that use Continuous Integration (CI) could also result in continuous noise (Poole 2008), which could make progress difficult, especially if CI was introduced during the later stages of the project. This is not, however, caused by CI, but exposed by it.

**Software testing levels**

The typical software testing levels, and how they relate to the system requirements, specification and design, are shown in Figure 1 (adapted from Wiegers 2003).

These testing levels are

- **unit testing**, informal testing used to verify the functionality of a software unit (e.g., a class). Testing is done in a “pure software” environment, that is, not on an embedded system. This testing is relatively easy, and so it is worth having extensive testing at this level to catch errors before more expensive testing at embedded or HILS test levels. An example of this is comparing a numerical algorithm’s output against reference results.

- **conformance testing**, informal or formal testing that verifies conformance of development...
Software testing tools

Static checkers

Static checkers evaluate source code; in these tests, the software is not executed.

The most basic such tool is the compiler: source code must be compilable in order to be executed; besides this, compilers give warnings for code that may be buggy. Static checkers can check conformance to coding standard rules (e.g., “every else-if chain must end with a bare else”), code metrics (e.g., McCabe cyclomatic complexity), and code style (e.g., whitespace
checkers). Examples of static analysis tools are PC-lint (Gimpel Software), Understand (Scientific Toolworks, Inc.) and Cppcheck (Daniel Marjamäki).

**Target emulators**

While it is useful to check implementations on development systems (typically Intel x86-class Windows systems), they must be tested on the final embedded target. A useful in-between step is to use a target platform emulator; this allows for testing of object code compiled for the final target on a development system.

An example of a target emulator is the “powerpc-eabi-run” command provided by the PowerPC cross-compilation suite (Sysprogs UG).

**Code coverage**

Code coverage tools let you check which parts of the product source code are executed during testing. This is usually fairly straightforward for code executed natively on development systems, but not always possible for embedded systems.

Microsoft Visual Studio (Microsoft) and gcc (Free Software Foundation, Inc.), both in widespread use, have code coverage capabilities.

**Test manager**

A test manager is a system that automates test execution; typically such a system will check source out of a repository, compile it, and execute a set of tests. This chain of events can be triggered by a repository check-in, or be done periodically. The manager can record the test binaries and results, and notify engineers in the event of failure.

Jenkins (Kohsuke Kawaguchi) is an open-source test management system; e.g., Figure 2 shows the Apache software foundation’s Jenkins CI server interface, which contains a list of test names and the status of each test.

**Test support libraries**

Test support libraries make it easier to write tests; they allow for organization of test suites, and for test conditions to be concisely and clearly expressed. Many such libraries are in so-called “xUnit” family, and variants exist for C, C++, C#, Java, Matlab, and Python (Wikipedia 2014).
Testing environments

Software-only testing

The software-only test environment tests the unit/system on a standard computer. It is flexible, since it can be used across most testing levels, from unit testing to system testing. This is typically an easily controlled environment, where the tests are repeatable.

Figure 3 shows the automated software-only test environment components:
• CI server: triggers tests for the Unit Under Test (UUT) and maintains a test time-line.

• Test controller: controls the execution of automated tests for the UUT, effectively substituting a human operator.

• Revision control server: allows UUT code check-outs.

• Compiler: compiles the UUT and reports compiler warnings and errors.

• Static code analyser: scans the UUT and reports style warnings and recommendations.

• Unit test/simulation: tests the UUT and reports test status (e.g., pass/fail).

• Issue tracking server: registers any newly revealed UUT issues.

As an example, consider the above applied the development and testing of the navigation system of a fictitious autonomous reconnaissance helicopter.

The navigation system is responsible for finding the position, velocity, and orientation of the helicopter, and of managing the related sensors (inertial motion sensors, satellite navigation receivers, etc.).

We assume that development has been going for some time. The navigation embedded developer now implements a new sensor calibration algorithm to be used before flight. She commits this implementation, together with a test set, to the project software repository.

The repository triggers the test manager to run through its software-only test suite. The test manager checks out the updated source code from the repository, and:

1. Compiles the new algorithm and tests for Windows x64 without warning or error.

2. Runs the tests; the newly committed tests pass, but an older test fails because it assumed sensor offset errors would not be calibrated out.

3. Code coverage analysis from the test execution reveal that the error-handling in the new code was not tested.

4. The code is then compiled for an emulated ARM target environment; several warnings are signalled by the cross-compiler.

5. The unit tests are run in an ARM emulator. Some of the new tests fail due to subtle differences in floating-point between the Intel and ARM math libraries.

6. A static analyzer detects that several new variables do not follow the project’s naming conventions.

7. The commit is marked as failing; the test manager opens a new issue on the tracking system, and sends an e-mail with a summary of the problems found to the erring developer.
Armed with this knowledge, she fixes the problems and makes a new commit that passes the software-only test suite.

The example illustrates that the developer must be able to execute the various tests outside of the test automation framework. If the testing queue is full, turnaround time for test execution will be long, and a developer might have started new work based on a substandard baseline by the time failure notification is sent.

**Stand-alone embedded testing**

![Diagram of automated embedded test environment](https://example.com/diagram.png)

Figure 4: Conceptual automated embedded test environment

The embedded test environment tests the stand-alone embedded system. It requires, at a minimum, a way to power the embedded system and also a way to communicate with it, e.g., an ARCNET (ARCNET Trade Association) interface. This is typically a controlled environment, where the test scenarios are repeatable.

Figure 4 shows the automated embedded test environment components:

- All the components described in Figure 3
- Power supply: powers the System Under Test (SUT).
- Emulator: emulates code execution on the SUT, without booting from non-volatile memory.
- Telemetry recorder: records SUT telemetry.
- Telemetry analyser: analyses recorded SUT telemetry to determine if test passes/fails.
Continuing the example from the previous section: the second commit passed the software-only tests, and so the test manager triggers the stand-alone embedded testing via a dedicated embedded tester.

1. The latest embedded source code is checked out, and cross-compiled warning-free into a test application to be loaded onto a single-processor ARM system.

2. The embedded tester powers on an attached ARM board, and uses this board’s boot-loader to load the new application.

3. The tester executes the stand-alone test suite: via a telemetry network, it sends test inputs to the ARM test application, and records the resulting outputs.

4. These outputs are compared to references, and are errors are within tolerance specifications in all cases. However, the algorithm execution time sometimes exceeds its time limit.

5. The commit is marked as failing. Another issue is opened, and an e-mail with details of the timing problem are sent to the developer. She commits a modified implementation that caches an often-used matrix multiplication result; this commit successfully pass through the battery of software-only tests, and is fast enough to pass the stand-alone embedded test.

Similarly to the software-only only example, this illustrates the need for the developer to run stand-alone embedded tests outside of the test automation framework. If test hardware is limited, this might mean stalling the test queue during debugging.

**HILS testing**

The HILS test environment tests the partially/completely integrated system in a simulated operational environment. It consists of interconnected embedded systems and simulators. This is typically a controlled environment, where the test scenarios are repeatable.

Figure 5 shows the automated HILS test environment components:

- Components described in Figure 3 and Figure 4.

- HILS: tests SUT in simulated operational environment.

- Programmer/loader: programs the embedded binaries to the non-volatile SUT memory.

Continuing our example further: after the calibration algorithm passes the stand-alone embedded testing, the test manager triggers the final stage of automated testing, using HILS. The test manager connects to the HILS controller and starts test execution.
1. The initial stages are similar to the stand-alone embedded testing: the latest embedded source code is checked out, compiled warning-free into the operational navigation application, and this application is loaded onto an ARM board. This board is integrated with boards of the mission controller, autopilot, and imaging systems.

2. The predetermined HILS scenarios are simulated. The new algorithm has no effect, however, since none of the scenarios was designed with calibration in mind, and do not allow enough pre-flight time for the calibration to converge.

3. The commit is marked as passing, and a notification mail is sent to the navigation developer. She belatedly realizes that a new scenario is needed to test the calibration, and contacts the HILS scenario developer. Together they adapt an existing scenario to include longer pre-flight time, and to have significant sensor biases that must be calibrated out.

4. The commit of this new scenario triggers a new HILS test run. The changed scenario fails because the autopilot was not using the calibration information provided by the navigation system, and performed poorly with the biased sensors. As with the other test setups, an issue is opened and notification mails are sent. The autopilot engineer commits a fix and regression test for the bug, which triggers a new cascade of autopilot software-only, stand-alone embedded, and HILS tests.

As with the other examples, this one shows the need to use the HILS outside of the test automation framework. Resource availability may be more of an issue than with stand-alone embedded testing, since HILS systems can be large and costly; there may be only one available for a given development programme.

It also illustrates that changes in one system may reveal problems in other systems, which may be problematic for concurrent development. To alleviate this, one could initially commit changes to a “staging branch”, and only allow the change through to the main line of development once all test stages have passed.
Operational testing

This test environment tests the completely integrated system in its expected operational environment, i.e. field testing. Problems found at this level could be very costly to resolve, since the test could be aborted in the event of a major problem, or the system could be damaged beyond repair. The uncontrollable nature of this environment (and the logistics involved with it) means that its testing process is typically not repeatable, and automated testing is infeasible.

Conclusions

Automated testing relieves developers and reviewers from the burden of routine testing. If used from early on in development, it allows for frequent, thorough testing; it can capture development history in the form of regression tests; and it can allow development to continue closer to deadlines.

Test automation can be used with unit test execution, including coverage checking, and static analysis. Tools exist to allow automation, to develop test suites, and to perform analysis of code and results.

Testing can be extended from typical “software-only” testing to executing in test frameworks on target embedded hardware, and to HILS testing. A simple example illustrates that developers also need to be able to execute tests outside the automation framework.

Acknowledgement

We thank Pieter van der Merwe for a series of discussions on simulation development he organized, at which the ideas presented here first arose.

Acronyms

CI Continuous Integration.

HILS Hardware in the Loop Simulation.

SUT System Under Test.

UUT Unit Under Test.
References


Pilgrim, Mark. 2004. *Dive into python*. APress Media, LLC.


Biographies

Reinhard Knoll graduated from the University of Pretoria in 2006 with an B. Eng. (Hons)(Computer Engineering). Since then he has been a radar software engineer at Denel Dynamics, and currently holds the position of principal software practitioner.

Rory Yorke graduated from the University of Cape Town in 2002 with an M. Sc. (Eng.). He has been a guidance and control engineer at Denel Dynamics for 13 years, and currently holds the position of chief software practitioner.