ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP053

TEST AUTOMATION FOR CONTROL SYSTEMS AT THE EUROPEAN SPALLATION SOURCE

Karl Vestin, Fabio dos Santos Alves, Lars Johansson, Stefano Pavinato, Kaj Rosengren, Marino Vojneski, European Spallation Source, Lund, Sweden

Abstract

This paper describes several control system test automation frameworks for the control systems at the European Spallation Source (ESS), a cutting-edge research facility that generates neutron beams for scientific experiments. The control system is a crucial component of ESS, responsible for regulating and monitoring the facility's complex machinery, including a proton accelerator, target station, and several neutron instruments.

The traditional approach to testing control systems largely relies on manual testing, which is time-consuming and error-prone. To enhance the testing process, several different test automation frameworks have been developed for various types of applications. Some of these frameworks are integrated with the ESS control system, enabling automated testing of new software releases and updates, as well as regression testing of existing functionality.

The paper provides an overview of various automation frameworks in use at ESS, including their architecture, tools, and development techniques. It discusses the benefits of the different frameworks, such as increased testing efficiency, improved software quality, and reduced testing costs. The paper concludes by outlining future development directions.

INTRODUCTION

The control system for a large research facility, such as the European Spallation Source (ESS), comprises thousands of different subsystems. Each of these subsystems has hundreds or even thousands of distinct configuration items, and each plays a role in fulfilling a specific function within the facility. The continued development and maintenance of such a complex system-of-systems present significant challenges in terms of verification.

The control systems at ESS are constructed using local control systems, typically based on Programmable Logic Controllers (PLCs) or high-speed data acquisition systems using Field Programmable Gate Arrays (FPGA)-based boards slotted into Micro Telecommunications Computing Architecture (MicroTCA) crates. All control systems are integrated into the unified control system using Experimental Physics and Industrial Control System (EPICS) Input Output Controllers (IOCs). All IOCs are based on reusable EPICS software modules distributed as part of the ESS EPICS Environment (e3) [1].

Manually testing and re-testing every component of the control system after each update is prohibitively time-consuming. To address this challenge, test automation is applied. Through the execution of test scripts in well-defined test environments, we can efficiently test our systems for regressions and faults before redeploying them after an update.

At ESS, various approaches to test automation are employed, tailored to the type and application of each specific control system. This paper will provide an overview of the diverse technologies and techniques used, discuss the rationale behind their selection, and present ideas for the future expansion of test automation.

EPICS MODULES

Most of the modules in the e3 environment are developed by the EPICS community and tested as part of the release process. ESS has developed dedicated test scripts to verify a subset of the modules. In most cases, the tests require some level of hardware emulation.

As an example, let us examine the unit test scripts for the Open Platform Communication Unified Architecture (OPCUA) e3 module [2]. The EPICS module is developed by the controls team at the International Thermonuclear Experimental Reactor (ITER) and wrapped by the ESS e3 team for integration into e3. All the unit tests are declared in a test script that is executed as a dedicated make target, essentially a defined objective for the general purpose tool for generation of executable from source code “make”. This make target is executed by a process - a runner - automatically started by the ESS GitLab continuous integration function upon check-in of the changes. Figure 1 illustrates how the results are stored in GitLab and are available for all historical builds.

Hardware emulation, in this case, is achieved using a dedicated OPCUA server based on the open-source code from the open62541 project [3]. The server is built as part of building the make target. The server is then started as part of the pytest test fixture in the test script to ensure the test cases always have a server to connect and test against.

One interesting aspect of the OPCUA unit test is the utilization of libfaketime [4], which is now installed by default on our test runner machines. Using this library, the test fixture can start the OPCUA server using a different ("fake") system clock, thereby verifying that timestamping is done correctly, as specified by the Time Stamp Event (TSE) field in the record.

Before a new version of the module is released into the production environment, the e3 team verifies that the pipeline runs cleanly, including any defined unit test scripts.

For the modules in the e3 environment, work is currently underway to improve test coverage through automated testing. The ambition is to have automated tests running as part of the continuous integration process.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP053

```
============================ test session starts =============================
platform linux -- Python 3.6.8, pytest-7.0.1, pluggy-1.0.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /builds/e3/wrappers/e3-opcua
collecting ... collected 36 items
test/test_opcua.py::TestConnectionTests::test_connect_disconnect PASSED  [  2%]
test/test_opcua.py::TestConnectionTests::test_connect_reconnect PASSED   [  5%]
test/test_opcua.py::TestConnectionTests::test_no_connection PASSED       [  8%]
test/test_opcua.py::TestConnectionTests::test_shutdown_on_ioc_reboot PASSED [ 11%]
test/test_opcua.py::TestVariableTests::test_server_status PASSED         [ 13%]
test/test_opcua.py::TestVariableTests::test_variable_pvget PASSED        [ 16%]
test/test_opcua.py::TestVariableTests::test_read_variable[VarCheckBool-True] PASSED [ 19%]
test/test_opcua.py::TestVariableTests::test_read_variable[VarCheckSByte--128] PASSED [ 22%]
test/test_opcua.py::TestVariableTests::test_read_variable[VarCheckByte-255] PASSED [ 25%]
test/test_opcua.py::TestVariableTests::test_read_variable[VarCheckInt16--32768] PASSED [ 27%]
test/test_opcua.py::TestVariableTests::test_read_variable[VarCheckUInt16-65535] PASSED [ 30%]
test/test_opcua.py::TestVariableTests::test_read_variable[VarCheckInt32--2147483648] PASSED [ 33%]
test/test_opcua.py::TestVariableTests::test_read_variable[VarCheckUInt32-4294967295] PASSED [ 36%]
test/test_opcua.py::TestVariableTests::test_read_variable[VarCheckInt64--1294967296] PASSED [ 38%]
test/test_opcua.py::TestVariableTests::test_read_variable[VarCheckUInt64-1.8446744073709552e+19] PASSED [ 41%]
test/test_opcua.py::TestVariableTests::test_read_variable[VarCheckFloat--0.0625] PASSED [ 44%]
test/test_opcua.py::TestVariableTests::test_read_variable[VarCheckDouble-0.002] PASSED [ 47%]
test/test_opcua.py::TestVariableTests::test_read_variable[VarCheckString-TestString01] PASSED [ 50%]
test/test_opcua.py::TestVariableTests::test_write_variable[VarCheckBool-False] PASSED [ 52%]
test/test_opcua.py::TestVariableTests::test_write_variable[VarCheckSByte-127] PASSED [ 55%]
test/test_opcua.py::TestVariableTests::test_write_variable[VarCheckByte-128] PASSED [ 58%]
test/test_opcua.py::TestVariableTests::test_write_variable[VarCheckInt16-32767] PASSED [ 61%]
test/test_opcua.py::TestVariableTests::test_write_variable[VarCheckUInt16-32768] PASSED [ 63%]
test/test_opcua.py::TestVariableTests::test_write_variable[VarCheckInt32-2147483647] PASSED [ 66%]
test/test_opcua.py::TestVariableTests::test_write_variable[VarCheckUInt32-2147483648] PASSED [ 69%]
test/test_opcua.py::TestVariableTests::test_write_variable[VarCheckInt64-0] PASSED [ 72%]
test/test_opcua.py::TestVariableTests::test_write_variable[VarCheckUInt64-0] PASSED [ 75%]
test/test_opcua.py::TestVariableTests::test_write_variable[VarCheckFloat--0.03125] PASSED [ 77%]
test/test_opcua.py::TestVariableTests::test_write_variable[VarCheckDouble--0.004] PASSED [ 80%]
test/test_opcua.py::TestVariableTests::test_write_variable[VarCheckString-ModifiedTestString] PASSED [ 83%]
test/test_opcua.py::TestVariableTests::test_timestamps PASSED            [ 86%]
test/test_opcua.py::TestPerformanceTests::test_write_performance XFAIL   [ 88%]
test/test_opcua.py::TestPerformanceTests::test_read_performance PASSED   [ 91%]
test/test_opcua.py::TestNegativeTests::test_no_server PASSED             [ 94%]
test/test_opcua.py::TestNegativeTests::test_bad_var_name PASSED          [ 97%]
test/test_opcua.py::TestNegativeTests::test_wrong_datatype PASSED        [100%]
=========================== short test summary info ============================
XFAIL test/test_opcua.py::TestPerformanceTests::test_write_performance
  GitLab runner performance issues
================== 35 passed, 1 xfailed in 334.98s (0:05:34) ===================
rm -f -rf /builds/e3/wrappers/e3-opcua/testMods-230612124651
Cleaning up project directory and file based variables

Job succeeded
```

Figure 1:Test for e3 module running in GitLab runner.

PLC DEVICE HANDLERS

Test automation for Programmable Logic Controllers (PLC) is implemented using a standardized Python virtual environment that packages the necessary tools for connecting to and testing a PLC project. All tests are currently executed on physical PLCs. Experiments have been performed to run tests on virtualized PLCs, but the results have so far not been satisfactory.

The tests are developed using pytest [5]. The script uses the Open Platform Communication Unified Architecture (OPCUA) to emulate input and output signals for the PLC and the EPICS Process Value Access (PVA) protocol to test the software interface to the PLC. Automation engineers at ESS use this environment to emulate inputs and outputs. In this way, they can verify the behaviour of the PLC software in a simple and repeatable fashion.

Currently, test automation is applied to the ESS device handlers (reusable functional blocks used across many different process automation projects). For example, the test script would run a comprehensive test suite for all the functions of a device handler for a control valve. This means that all process automation projects can rely on this retested component to be stable and well-tested.

For verification of PLC-based control systems, further investigation is ongoing into how simulated PLC hardware could be used to allow testing to be integrated into a continuous integration workflow without reliance on physical hardware. Additionally, modelling and testing against simulated processes to leverage the test methodology from component testing to system testing are under discussion.

FIRMWARE VERIFICATION

FPGAs using Very High-Speed Integrated Circuit Program (VHSIC) Hardware Description Language (VHDL) are used in several systems at ESS, including but not limited to:

- The ESS Fast Beam Interlock System (FBIS).
- Data acquisition for beam diagnostics
- Neutron detectors
- Radio frequency (RF) systems

Data Acquisition Systems

Functions implemented by the firmware are verified using the Universal Verification Methodology (UVM) [6] test benches in Questa Advanced Simulator [7]. These methods generate constraint-randomized test vectors to allow efficient testing at the function level.

Multi-unit and chip-level verification are carried out using hand-written functional test cases and scenarios developed in System Verilog [8] and simulated using Questa Advanced Simulator.

For system-level testing (in this case, testing the firmware as it runs on an FPGA board), development is done in Python and pytest and LowLevHW, a Python library for handling hardware developed at ESS [9]. For example, emulated Analog-to-Digital Converter (ADC) data can be injected for filter chain testing. These tests also include performance and stability test cases, and the test reports are automatically uploaded to a common file share for traceability.

Ongoing development focuses on requirement-driven firmware unit testing using cocotb [10] and/or pyuvm [11]-based approaches. These methods offer an efficient and standardized way to develop test benches using a scripting language, Python, instead of Register Transfer Languages (RTL) like Verilog.

The ultimate objective is to create a common and reusable framework and library for testing both at the firmware and system levels for FPGA-based systems.

Fast Beam Interlock System

The ESS Fast Beam Interlock System (FBIS) was developed in collaboration with ZHAW [12]. A comprehensive Hardware-In-the-Loop (HIL) test framework based on a National Instruments test rig was provided. Work is ongoing to build an updated and refactored test automation framework based on an FPGA-based test rig and test scripts in pytest to improve efficiency and maintainability.

The new test suite currently consists of approximately 1000 test cases, providing test coverage similar to that of the originally provided test framework.

Verification results are stored in the ESS document management system and used as a reference in the system test report. The verification reports are under version control and are part of the ESS facility baseline.

Neutron Detectors

The ESS Readout Master Module was designed in collaboration with the Science and Technology Facilities

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP053

Council (STFC) as a generic readout solution for various neutron detector electronics that interface with data using FPGAs. The primary function of the ESS Readout Master is to simplify communication between the FPGAs and other systems by serving as a point of aggregation. This enables the establishment of a common and standardized interface. The interfacing systems include the integrated control system and timing provided by the Integrated Controls Group (ICS), as well as event formation provided by the ESS Data Management and Software Centre (DMSC).

The current development of the firmware is verified using the VUnit framework [13] through test benches in Questa Advanced Simulator [7]. After function testing using the simulator, the next step is HIL verification. This involves connecting the System Under Test (SUT) to an operational model that mimics the real-world conditions in which the ESS Readout Master will operate.

BEAM DIAGNOSTICS INTRUMENTS

Figure 2 outlines the procedure for verifying firmware applied to the ESS Beam Current Monitor (BCM). After the firmware has been verified, the system functionality is tested on hardware in the lab. The device is controlled and monitored with an EPICS IOC during the test.

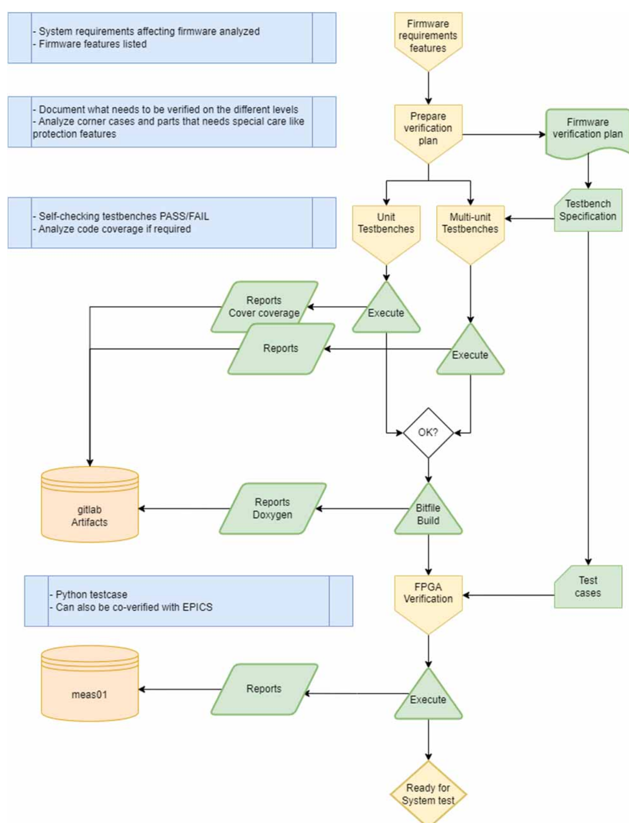

Figure 2: Firmware verification flow.

Both test stimulus and results are managed through the EPICS PVs, which provide a high degree of flexibility regarding tools for running the test scripts. Currently, we have test automation projects primarily using WeTest [14] and MATLAB [15].

The results from the tests are stored in a shared file storage system to enable traceability of test results for any deployed version of the instrument's firmware and software.

Looking ahead, our ambition is to use pytest for test scripting due to its relative simplicity and to harmonize with other test automation efforts at ESS.

APPLICATION SOFTWARE

The software team at ESS develops and maintains a suite of application software, including EPICS channel finder, naming service, cable database, controls configuration database, online logbook, archiver appliance, and control system studio. Many of these applications are developed in the EPICS community, and ESS contributions are handled through upstream pull requests, following the quality assurance guidelines of the upstream repositories. This often includes test scripts for test automation.

For in-house development, such as the naming service, and some community applications, such as channel finder, automated unit tests are developed using JUnit 5, test containers [16], and Java code. The usage of containers for managing dependencies is essential to ensure stable and repeatable test results.

The strategy for developing test automation focuses on providing functional testing of central parts of the software. As an example; there are rules for how users can set the names in the naming service. Since many other tools depend on these names to function correctly, it is important that this functionality has substantial test coverage.

Integration test automation is employed at the application level for functional verification. The tests run on the representational state transfer application programming interface for the application to ensure repeatable test results without relying on user interface interactions. A test library (ITUtil) with helper methods for integration testing has been developed to facilitate testing.

An example of a simplified test sequence could include:

1. Create an application instance with an empty database
2. Verify that the application is running and accessible
3. Add data items, one by and one or in batches
4. Perform various actions (e.g. add, update, delete and verify)
5. Verify that each action has the expected result

The results from automated testing are used to assess the maturity and stability of the application before release into the production environment.

Looking ahead, the ambition for test automation for application software is to gradually expand test coverage and investigate the possibility of integrating test automation into the continuous integration environment.

CONCLUSION

Test automation is implemented across multiple disciplines at ESS, primarily to ensure that we can update and maintain our systems without significant risks of causing regressions in existing functionality or introducing new faults.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP053

Data generated by the tests is stored as required to ensure that we can trace back to the test results if we ever encounter problems in the production environment.

The current trend is to expand testing to cover more areas, increase test coverage, and, simultaneously, harmonizing testing methodologies and tools across various domains. The direction of technology choice is moving towards standardizing on Python-based test automation for most disciplines.

ACKNOWLEDGEMENTS

We would like to extend our gratitude to the team at Zurich University of Applied Sciences (ZHAW) for their invaluable contributions to the development of the Fast Beam Interlock FPGA system and the test rig.

We also wish to express our sincere thanks to Ralph Lange and his team at the International Thermonuclear Experimental Reactor (ITER) for their exceptional work in developing the EPICS OPCUA module and their ongoing support.

Finally, we would like to express our gratitude to the team at STFC for their invaluable contribution to the ESS readout master module for neutron detectors.

REFERENCES

[1] ESS EPICS Environment, https://gitlab.esss.lu.se/e3

[2] e3-opcua, https://gitlab.esss.lu.se/e3/wrappers/e3-opcua

[3] open6241, https://www.open62541.org

[4] libfaketime, https://github.com/wolfcw/libfaketime

[5] pytest, https://docs.pytest.org/en/7.4.x/index.html

[6] Universal Verification Methodology, https://enQuesta .wikipedia.org/wiki/Universal_Verification_Methodology

[7] Questa advanced simulator, https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/

[8] SystemVerilo, https://en.wikipedia.org/wiki/SystemVerilog

[9] LowLevHW, https://gitlab.esss.lu.se/fpga/lowlevhw

[10] cocotb, https://www.cocotb.org

[11] pyuvm, https://github.com/pyuvm/pyuvm

[12] WeTest, https://www.wetest.net

[15] MathWorks, https://se.mathworks.com

[16] Testcontainers, https://testcontainers.org

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI