ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA090

TESTING TOOLS FOR THE IBEX CONTROL SYSTEM

T. Löhnert, F. A. Akeroyd, K. Baker, D. Keymer, A. J. Long, C. Moreton-Smith, D. Oram,
ISIS Neutron and Muon Source, Didcot, UK
J. R. Holt, T. A. Willemsen, K. J. Woods, Tessella, Abingdon, UK

Abstract

At the ISIS Neutron and Muon Source [1], we are currently in the process of replacing the legacy SECI control system with the next generation IBEX control system [2] based on EPICS [3]. Since IBEX replaces a fully functional control program, users and developers need to have confidence that the new system works as well as the old one, ensuring that the migration does not cause any undue disruption to beamline operations. Automated testing is an indispensable tool to continuously ensure our software is up to the highest standard of quality. This paper gives an overview of the various types of testing tools we utilize at ISIS, with a particular focus on our IOC test framework [4]. This framework has been developed in-house for testing drivers with simulated devices in order to circumvent testing limitations due to beamline/device availability.

INTRODUCTION

For the last 5 years, we in the Experiment Control team have been developing the next-generation instrument control system called IBEX for use on beamlines in the ISIS facility. The migration to IBEX is an ongoing process, with around half of our beamlines currently running on the new system. Being such a large, complex and long-running project make IBEX prone to failure if proper care is not taken. Consider the following facts:

- IBEX is a complex ecosystem consisting of many different services. It is not always obvious what impact changes to one component will have on another
- Over the course of its lifetime, over 30 developers have contributed to the IBEX project, many only for a limited duration and sometimes early in their software engineering careers
- The IBEX system is too complex even for permanent staff to be familiar with every part of it at any one time, let alone have in-depth knowledge of it
- Devices are often fixed on the beamline they are used on. Since beam time is highly valuable, it is often difficult to get sufficient time to test a new device driver before it is needed in production. This can similarly apply to re-testing if changes/updates are later made

Automated testing helps meet all of the above challenges. A robust set of tests ensures any unforeseen changes in behaviour are caught before they are deployed to production beamlines. Once written, robust tests benchmark any newly developed code to forever protect it from regression. Tests can also act as a form of documentation for unknown parts of the system (in addition to other forms of documentation), as they demonstrate the expected behaviour of the component being tested, which is especially helpful if the original author has since moved on. Finally, having a well-designed test framework with strong simulation capabilities goes a long way to ensure the surprises when testing with a real beamline are kept to a minimum.

Robust testing practices raise the confidence of the developers that new code works as intended, and that existing code continues to work when subjected to changes. It also helps raise the confidence of users, who are generally risk-averse that the new control system they are being given performs its function satisfactorily.

In the following sections, we will explore a selection of relevant IBEX components, and the tools in place to ensure their continued functionality.

IBEX COMPONENTS

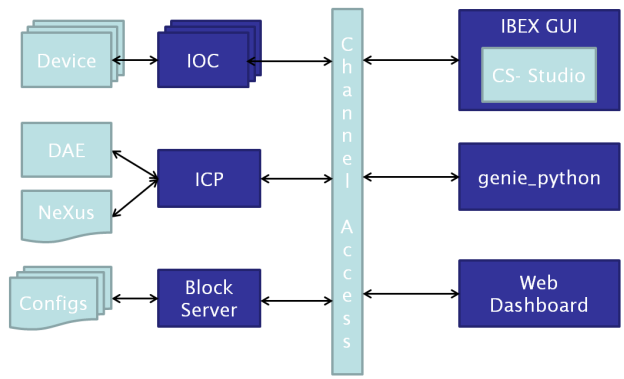

Figure 1: Overview of (selected) IBEX components.

IBEX covers a wide range of responsibilities related to experiment control, from interacting with individual bits of beamline equipment such as temperature or pressure controllers, choppers, jaws etc. to managing data collection and writing the final experiment data file. To this end, it comprises a number of technologies and services.

Figure 1 shows a schematic of the core system. On the left side, we have the server comprised of various backend components:

- A number of devices which are driven through EPICS Input-Output Controllers (*IOCs*), which expose relevant values to be read/written to over the network using the EPICS Channel Access protocol.
- The Instrument Control Program (ICP), which deals with the neutron data (collection, processing and writing to file) obtained from the Data Acquisition Electronics (DAE)
- The Block Server, which is a python process that manages the beamline configuration – i.e. which devices to control, which sample environment values to log etc.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA090

On the right hand side, we have various frontend clients that communicate with the server through which users can interact with the system:

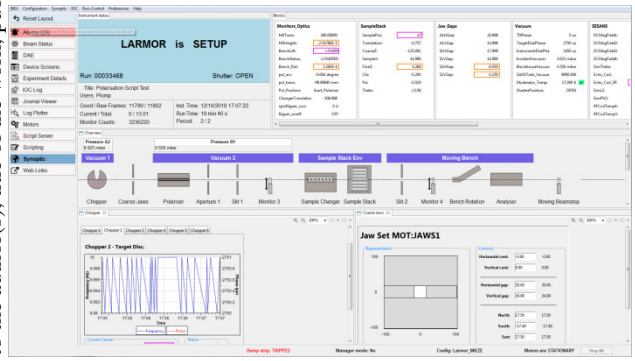

Figure 2: The IBEX GUI.

- The IBEX GUI – this is an Eclipse/RCP desktop application based on the CS Studio framework [5], and represents the primary form of interaction (shown in Fig. 2)
- Genie_Python [6], a library of python commands used to interact with the control system via a command line interface or through running user-written scripts
- The web dashboard, a basic web interface, which provides a read-only view of the beamline status.

Note that there are many other services and features that are not included in this selection, for example those that manage device alarms or logging. However, this representative set of core components is useful for illustrating our use of various different forms of automated testing.

UNIT TESTING

Unit tests are a standard software engineering tool most developers should already be familiar with. They are used to test the system "bottom up" by testing the behaviour of small pieces of code (e.g. individual methods) in isolation, and should complete in a matter of seconds or less. Each unit test should be strongly focused and mock (simulate) functionality outside of that which is being tested, e.g. a test for a method parsing the content of a text file should never fail on account of a file system error.

Most of the languages used in our code base come with unit testing frameworks (e.g. JUnit in Java, unittest in Python). We are conscientious that the code we write is structured in a way that permits comprehensive testing. We do this for example by using test driven development [7], a coding approach where one first writes tests, and then the code to make them pass, or by conventionally using a Model-View-Viewmodel pattern [8] for GUI code, which makes display logic easier to test by separating it from pure views.

Furthermore, all our unit tests follow certain conventions in terms of test naming and structure. This is helpful for our developers when trying to understand a previously unknown part of the system, and thus the tests themselves serve as a form of documentation.

It has been our policy for several years now to not deploy any new code that does not come with a reasonable set of tests. As such, out of the components above, the GUI client, Genie_Python, web dashboard and blockserver components all come with substantial unit test suites.

Some older parts of the system (such as the ICP) are missing unit tests because they are rarely touched and we generally consider them stable, which means implementing unit tests has not become a sufficiently high priority. In the meantime, we use static code analysis tools to keep track of test coverage so that we remain aware of such gaps.

For IOCs, we do not have what would strictly be considered unit tests, since they require at least a device to test against. Devices can be arbitrarily complicated, thus mocking their behaviour is a bit more involved.

DEVICE TESTING

A large part of the work we do is writing IOC drivers for specific devices used on the beamlines. EPICS provides meta-languages which make it easy to implement device drivers, but these are bespoke and do not come with any testing frameworks. Thus, we have developed an in-house IOC testing framework which allows you to write tests in python that test the behaviour of IOCs against the devices with which they communicate.

Devices are often fixed equipment, meaning we cannot confirm our drivers behave correctly without having access to the beamline, which is not always easy or indeed possible. With smaller devices, we are sometimes able to test against the device in the office, but even so, we usually only have these available for a limited time and therefore they are unsuitable for use in a continually running automated test suite.

Because of this, we simulate the behaviour on the device side, which we can do at two different levels: The EPICS record level, and the device level.

Record Simulation

Record simulation is a concept that is built into the EPICS framework [9]. Every one of our IOCs has a "simulate" field, which, when enabled, will use a set of aliased dummy simulation database records instead of the real ones. These simulated records bypass communication with a real device and instead use purely virtual values held in the IOC itself. While it is possible to link simulation records logically (e.g. when setting a setpoint on a parameter, automatically update its readback value, too), the EPICS database language is too restrictive to effectively simulate complex device behaviour. Still, record simulation is useful for confirming basic IOC behaviour without the need to communicate with a device.

Device Simulation

Device emulators allow us to simulate arbitrarily complex devices. Most of our emulators are written using the LeWIS package [10]. LeWIS lets you build complex stateful device simulations in python, emulates a variety of protocols and provides backdoor access to these emulators to simulate external events such as moving into an error state after a sensor has been disconnected. Note that

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA090

LeWIS does not depend on the IOC test framework, and neither does the test framework require emulators to be written in LeWIS – for instance, we also test drivers for our new Beckhoff [11] motion controllers against the vendor-provided emulator.

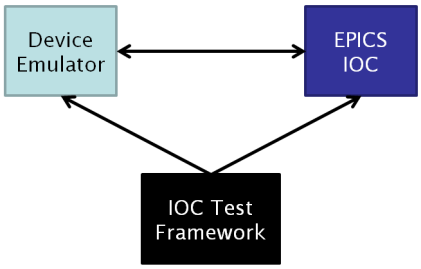

Figure 3: Schematic of the interaction between emulators, IOCs and the test framework.

Test Case Example

Following is an example of a test, which checks that disconnecting the sensor of a temperature controller stops it ramping. Figure 3 shows the relationship between the components involved.

1. We start the IOC test suite for a temperature controller. It automatically runs up the relevant device emulator and IOC as defined in the test class. In this case we only run a single device but test classes may contain multiple IOCs / emulators if necessary, e.g. if we want to test the interaction between them
2. Once it has confirmed the emulator and IOC have successfully started up, the framework begins running the test case.
3. As part of the test case, it instructs the IOC to start ramping up the temperature via channel access
4. It checks that the emulated device has gone into a "ramping state"
5. The test procedure now tells the emulator via the backdoor to act as if the sensor has been disconnected
6. Finally, we assert with both LeWIS and the IOC that the device has stopped ramping.
7. Once completed, the test framework runs the test suite again in record simulation mode (if applicable). We can specify for each test whether we want to run it in record or device simulation mode, or both.

At the time of writing, we have emulators for around 60 different devices. Depending on the device, some have comprehensive test suites with dozens of test cases, while some may simply check that an IOC successfully boots and can talk to the emulator. One may think that once a driver has been written and confirmed to work, there is no need to repeatedly run these tests, however in reality they have frequently flagged up broken functionality, e.g. when a library an IOC depends on has been changed.

Another use of the IOC test framework is to reverse engineer drivers from the old SECI experiment control system. Starting with the old driver, we write an emulator that behaves as that driver expects. We can then write and validate an IOC against the emulator created in this way. This is useful as we sometimes have drivers for old equipment under SECI for which no manuals exist, or whose manufacturers do not operate any more.

It should be said that there is no replacement for testing an IOC with the real device as unforeseen things can always happen (e.g. mistakes in a device manual on which we base emulator behaviour). However, the IOC test framework still lets us dramatically reduce the risk associated with using a new or changed device driver in a production environment.

GUI/SYSTEM TESTING

Another system component that requires testing is the graphical user interface of the main IBEX client. A large portion of the features in the GUI provide access to and rely on functionality found in the IBEX backend. Therefore, testing the GUI behaves in the correct way in most cases means testing the entire IBEX system behaves in the correct way, and thus GUI tests often double as system tests.

Testing the user interface requires input by a user (i.e. mouse clicking, entering text/values). Thus, originally, much of our system testing was performed manually by developers. Running through this set of 100+ tests for a new release proved to be by far the slowest part of deploying a new version of IBEX. Because of this, today we use an automated GUI testing tool called Squish [12]. It is worth noting that Squish is not free software, however after surveying various available tools, we found it to be the best option for a number of reasons:

- Squish is easy to use in terms of its API and by providing functionality to generate test cases from recorded user interaction
- Test cases can be written in python, a language most of our developers are already familiar with
- Squish is highly extensible as tests interact with the application on the OS level, and are therefore independent of the language in which the UI is written

Automating these tests mean they can run frequently without requiring any intervention by the developers. This gives us confidence that functionality across the entire IBEX stack is working correctly at any given moment.

In addition to speeding up releases and saving developer time, this tool has helped us find bugs which require testing that would not be feasible or at least extremely tedious to perform manually, such as repeatedly going through a specific workflow to trigger a rare race condition or to investigate the source of a memory leak.

We also have another set of system tests, which test the IBEX stack by executing Genie_Python commands instead of interacting with it through the GUI. Both cases are again aided by simulation – since we do not have real live neutron data available when running these tests, they are run with a simulated version of the data acquisition component, which provides dummy data.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA090

CONTINUOUS INTEGRATION

For all the virtues of automated tests, they are only useful if they are run frequently. We have set up a continuous integration system using Jenkins [13], which runs our various test suites on one of our dedicated build servers either stand-alone or in the case of unit tests, as part of the build of the component they are testing.

Figure 4: The IBEX build status board.

The build for a component is run every time it is modified, and additionally once every night. We have a status board for all the builds in the IBEX project (see Fig. 4), which is displayed on a screen in our office so that we as a team may monitor it constantly. We also explicitly look at the board as part of our daily stand-up routine, to discover when and why builds are breaking, so that we may fix the issues as soon as possible.

Other Utilities

Continuous integration is not limited to running conventional tests on a codebase somewhere. For example, we use Jenkins to run a number of sanity checks on other parts of our system, such as:

- Configuration checks: The configuration checker runs a set of validity tests on beamline configurations, and has helped us find issues such as invalid IOC settings meaning the driver would not have run properly, or duplicate items in configurations which could have resulted in unpredictable behaviour
- Repository checks: These make sure that our git repositories are properly sanitised, and have caught errors in the past where submodules were not properly linked, meaning new code would have failed to be deployed
- Wiki Checks: These tests help ensure our online documentation is correct, e.g. by checking words are spelled correctly or URLs contained within are valid

FUTURE WORK

Developing robust tests is equally as important as developing the IBEX codebase itself, and as such we are continuously striving to improve our testing infrastructure. A lot of the processes and tools described in this paper have been introduced over the course of the project as a way to manage arising issues by improving maintainability and robustness. While new or changed code written for the project today is generally supported by solid testing, coverage is still far from perfect, particularly in older parts of the system. We use tools to analyse code coverage in order to identify parts of the system that require work in this regard. Improving our unit test coverage is one of the most straight-forward ways we have to increase robustness of the whole system.

Further, we still have some system tests that are yet to be automated. This is due to limitations on available effort and having to balance this work with other tasks, taking into account the risk and effort associated with running these tests manually only. However, we do occasionally dedicate a day of development to reducing technical debt in a specific area of the project – automating the remaining manual system tests is one potential subject for this activity.

Another arena in which we would like to expand testing capabilities is user scripts. These are scripts written using the Genie_Python library, not by the development team, but by facility users in order to automate their experiments. Inevitably, mistakes sometimes creep into these scripts, which can on occasion lead to lost beam time, e.g. an experiment is being run over night with the wrong parameters. The avenue we are exploring to prevent such occurrences is by providing a "dry run" option for user scripts, which simulates the actions in the script (e.g. just printing them out instead of performing them), which only takes a few moments after which users can confirm their correctness before committing to running the actual script.

CONCLUSION

To summarize, automated testing at various levels has held a multitude of benefits to the long-term health of the IBEX project:

- It helps reduce lost beam time as drivers are tested before being used in production, letting developers resolve issues before they arise
- It speeds up the release cycle as time spent on manual system testing is kept to a minimum.
- Regularly running the full test suite increases confidence in the system
- It prevents regression as broken functionality becomes visible immediately
- It helps document the code as tests demonstrate what behaviour is expected
- Testing methodology can be extended to sanitize other parts of the project beyond purely code

REFERENCES

[1] ISIS Pulsed Neutron and Muon Source, http://www.isis.stfc.ac.uk

[2] K. V. L. Baker *et al.*, "IBEX: Beamline Control at ISIS Pulsed Neutron and Muon Source", presented at the 17th Int. Conf. on Accelerator and Large Experimental Control Systems (ICALEPCS'19), New York, NY, USA, Oct. 2019, paper MOCPL01, this conference.

[3] EPICS Control System Framework, http://www.epics-controls.org

[4] IOC Test Framework, http://www.github.com/ISISComputingGroup/EPICS-IOC_Test_Framework

[5] Control System Studio, http://www.controlsystemstudio.org

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA090

[6] Genie_Python, http://www.github.com/ISISComputingGroup/genie_python

[7] Beck, Kent. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[8] Garofalo, Raffaele. "Building enterprise applications with Windows Presentation Foundation and the model view ViewModel Pattern". Microsoft Press, 2011.

[9] Wright, R. M., D. M. Kerstiens, G. D. Vaughn, and R. E. Weiss. "EPICS simulation tools for control system development." No. LA-UR-94-2781; CONF-9408125-18. Los Alamos National Lab., NM (United States), 1994

[10] LeWIS – Let's Write Intricate Simulators, http://www.github.com/ess-dmsc/lewis

[11] Beckhoff, http://www.beckhoff.co.uk

[12] Squish, http://www.froglogic.com/squish

[13] Jenkins, http://www.jenkins.io