# TESTING THE UNTESTABLE: A REALISTIC VISION OF FEARLESSLY TESTING (ALMOST) EVERY SINGLE ACCELERATOR COMPONENT WITHOUT BEAM AND CONTINUOUS DEPLOYMENT THEREOF

A. Calia, K. Fuchsberger, M. Hostettler, CERN, Geneva, Switzerland

*Abstract*

Whenever a bug in some piece of software or hardware stops beam operation, loss of time is rarely negligible and the cost (either in lost luminosity or real financial one) might be significant. Optimization of the accelerator availability is a strong motivation to avoid such kind of issues. Still, even at large accelerator labs like CERN, release cycles of many accelerator components are managed in a "deploy and pray" manner. In this paper we will give a short general overview on testing strategies used commonly in software development projects and illustrate their application on accelerator components, both hardware and software. Finally, several examples of CERN systems will be shown on which these techniques were or will be applied (LHC Beam-Based Feedbacks and LHC Luminosity Server) and describe why it is worth doing so.

## INTRODUCTION

An accelerator is a complex system, consisting of many interlinked components, which are typically organized in a control system of different layers from top-level applications to actual hardware.

Fig. 1 shows a vertical slice of a typical accelerator control system stack: On top there is the application layer, consisting of a set of physics-aware applications used by operators, which accesses the hardware through a middle layer. Below, the hardware layer is responsible of actually driving the hardware interacting with the beam.

An accelerator component is typically on one of these layers, and accessing or being accessed by one or more neighbouring components from the other layers. E.g. a top-level software component can access different hardware components through the middle layer, while a hardware component is often accessed by different top-level applications.

## EXECUTION MODE

For testability of individual components or any subset of the whole control system, it is required to reduce the coupling between neighbouring components. To facilitate decoupling, we propose different "execution modes" for an accelerator component, which can make it independent of the input and output of other components for testing and development purposes.

### *Simulation*

In a simulation mode, the component's inputs are based on a model. This model is dynamic and can be affected by the execution of a component. The purpose of a simulated model is to test the component in a dynamic environment that can be close to the reality (production).

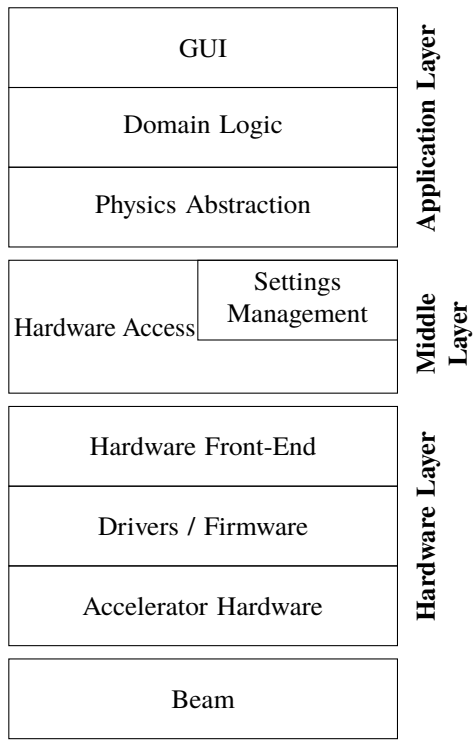



Figure 1: Vertical slice of a typical control system.

Ideally, it is possible to create various simulation models to effectively test the component under different circumstances. For a web service, a particularly interesting test is to verify the behaviour when network communications are very unstable and randomly slow. In the case of a hardware component, a challenging model can produce random noise in the inputs signals of the hardware cards.

### *Scenario*

A scenario is composed by a set of fixed values that are the inputs of a component. Given the scenario's input, it is possible to assert the component's output to spot errors.

Scenarios can be created from particular situations that the component must be able to handle. These situations can be artificially made, based on the component's design, or can be derived from experience. The latter case is especially true in a high-availability system. In these kind of systems it is precious to not introduce regressions during updates.

ISBN 978-3-95450-177-9



Copyright © 2016 CC-BY-3.0 and by the respective authors

### *Production*

In production mode, the component is executed without any restriction and it is not aware of the ongoing test. In this context, all the components in production mode must be able to fully access hardware and low-level systems.

For security reasons, the only part that can differ from the real operational mode is the environment. If the component (or components) to test are software, the environment is allowed to be a sand-box in which a failure is not propagated to the real operational environment. For a hardware component, a testing facility that replicates the real environment is highly advised.

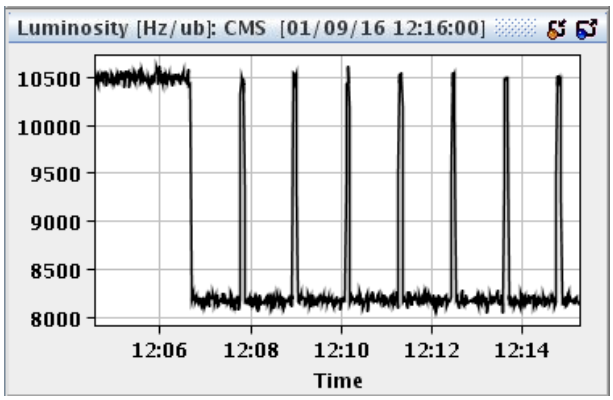



Figure 2: LHC Luminosity Server in full simulation mode, showing a luminosity plot while running a scan pattern used for luminosity calibration. Since the server runs in simulation mode, no accelerator controls are accessed and the response (beam displacements and resulting luminosity) is simulated.

## TESTING STRATEGIES

### *Unit Tests*

Unit Tests test each subcomponent, e.g. a class in a software project or a part of a hardware device, individually against a predefined set of scenarios and expected results.

Ideally unit tests should be implemented in an automated fashion on every layer of the system to quickly spot and pinpoint most breaking changes during development. No communication between components are involved in such tests.

### *Single-Component Integration Tests*

Single-Component Integration Tests shall ensure proper linking and communication between the individual subcomponents, i.e. that the full component behaves correctly in a particular scenario. While the subcomponents of the component being tested should be linked as they are in production, all communication to other components should be mocked.

Such tests can either run automatically against a predefined set of scenarios and expected responses, or manually with the component in simulation mode.

### *Integration Tests with Other Components*

Once a component is found to be working, proper communication to neighbouring components needs to be tested. To allow this, two or more components are linked together, while all components not involved in the tests should be mocked up.

Such a test could e.g. ensure that a control application can communicate to the hardware access layer, without actually driving hardware. On the other hand, it could also allow testing a hardware, it's drivers and front-end layer without beam, and without the upper layers of the control system.

During the testing, the scope of tests can gradually be extended by involving more components, up to the final commissioning with beam.

### *Sanity Checks in Operation*

To ensure reliable operation over an extended period of time without degradation, operational hardware can run through a set of unit or single-component integration type tests in every machine cycle. E.g. at the LHC, the Beam Loss Monitors (BLM) execute a set of "sanity checks" when preparing the machine for injection. [1]

## ERROR HANDLING AND LOGGING

Once an accelerator component has been deployed, commissioned and put in operation, it is crucial to detect problems and report them to the operators. The worst case is a component silently failing without any notice or explanation.

In case of a software component, this requires that possible errors and exceptions are properly handled, logged and possibly displayed to the user. If the error is recoverable from but could possibly affect further operations, a warning should be issued to the user. If the software component can't recover from a particular error, it must fail in a well-defined way, providing any available information on what problem occurred to the user.

However, care must be taken not to raise false warnings, as this leads to the warnings being ignored by the users. For later offline analysis by the developers, a verbose debug log and telemetry data can be collected through a central logging and tracing service.

## CONTINUOUS INTEGRATION

Continuous Integration (CI) refers to the ability of continuously build and integrate software. This means run tests with the latest version of each dependency to have finer integration. With this approach it is possible to immediately spot errors and regressions since the changes between each test run are small, possibly just a commit.

The CI process is often delegated to a CI server (Fig. 3) that is also able to deploy a SNAPSHOT (development) version of the software.

The deployment to production is a critical step since the software may be used in operations, for example during Physics Collisions in the LHC. Therefore, a Continuous

Copyright © 2016 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-177-9

Deployment (CD) approach cannot be done in a automatic way.

Nevertheless, the CI server can build and test the new version of the software and set it ready for deployment. Then, whenever the machine operator decides it is safe to deploy, the CI server perform the deployment. This step should be fully automatized and during the process all the running instances of the software should be automatically restarted and the links to run them updated.

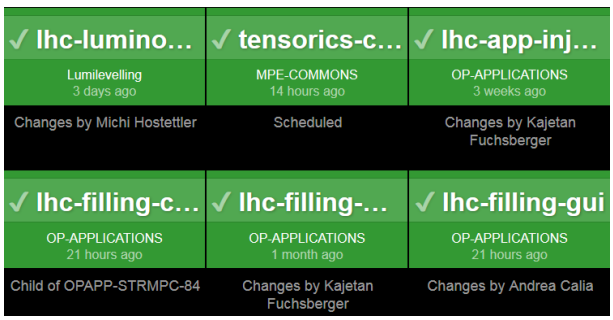



Figure 3: CI server dashboard. It is used to keep track of regressions introduced by commits on key projects. Each time a project is updates, the corresponding and dependant tests are run.

## EXAMPLES

### *LHC Luminosity Server*

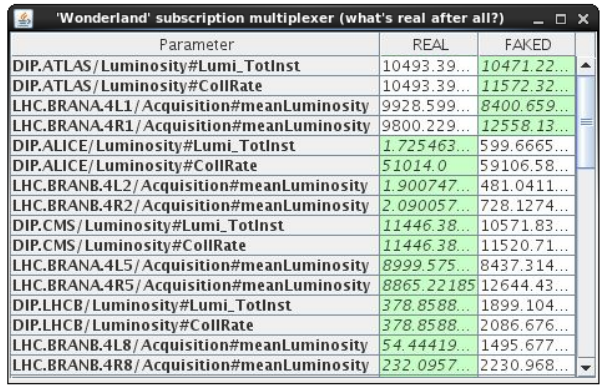
The LHC Luminosity Server is used to control the beams at the LHC collision points. In routine operation, it is mostly used to perform automatic scans, displacing the beams slightly against each other, to find the optimal head-on collision point. For absolute calibration of the luminosity monitors of the LHC experiments using the Van-der-Meer method [2], it can also run arbitrary scan patterns as requested by the experiments.

The Luminosity Server features a built-in simulation mode, which allows to develop and test scan strategies and patterns without accessing the middle layer at all. Hence every developer can fearlessly start their own instance of the server locally for testing and development, without interfering with other developers or even beam operation.

For integration tests with other components, the Luminosity Server also provides partial simulation (Fig. 4), communicating with some neighbouring components while simulating others. This is e.g. used to test the communication with an LHC experiment regardless of and without affecting any LHC beam operation in parallel, which saved several hours of beam time in 2016 during the preparation of the luminosity calibration sessions.

### *LHC Beam-Based Feedbacks*

A system which could be tested only with beam for a long time, is the LHC beam based feedback system. It processes input from about 2000 devices (orbit, tune), calculates corrections at a rate of 12.5 Hz and sends out corrections to about 500 correction magnets.




| Parameter | REAL | FAKED |
|---|---|---|
| DIP.ATLAS/Luminosity#Lumi_TotInst | 10493.39... | 10471.22... |
| DIP.ATLAS/Luminosity#CollRate | 10493.39... | 11572.32... |
| LHC.BRANA.4L1/Acquisition#meanLuminosity | 9928.599... | 8400.659... |
| LHC.BRANA.4R1/Acquisition#meanLuminosity | 9800.229... | 12558.13... |
| DIP.ALICE/Luminosity#Lumi_TotInst | 1.725463... | 599.6665... |
| DIP.ALICE/Luminosity#CollRate | 51014.0 | 59106.58... |
| LHC.BRANB.4L2/Acquisition#meanLuminosity | 1.900747... | 481.0411... |
| LHC.BRANB.4R2/Acquisition#meanLuminosity | 2.090057... | 728.1274... |
| DIP.CMS/Luminosity#Lumi_TotInst | 11446.38... | 10571.83... |
| DIP.CMS/Luminosity#CollRate | 11446.38... | 11520.71... |
| LHC.BRANA.4L5/Acquisition#meanLuminosity | 8999.575... | 8437.314... |
| LHC.BRANA.4R5/Acquisition#meanLuminosity | 8865.22185 | 12644.43... |
| DIP.LHCB/Luminosity#Lumi_TotInst | 378.8588... | 1899.104... |
| DIP.LHCB/Luminosity#CollRate | 378.8588... | 2086.676... |
| LHC.BRANB.4L8/Acquisition#meanLuminosity | 54.44419... | 1495.677... |
| LHC.BRANB.4R8/Acquisition#meanLuminosity | 232.0957... | 2230.968... |

Figure 4: LHC Luminosity Server in partial simulation mode. Here, e.g. the luminosities provided by the LHC experiments can either be picked up, or replaced by simulated values. The GUI provides a real-time view of both the real value and the output of the simulator, and the data source can be changed at any time.

For a very long time it was considered too complicated to write a testing framework for this system. Finally, when in 2015 an attempt was started anyhow, it proved to be less complicated than expected. This is a very good example which shows that with clever slicing of a system and starting with simple tests, practically any system can be tested. Part of the "fear" before starting to create the testing system was based on the assumption that a full simulation framework would have to be created, which would have to be as fast as the feedback system. This of course would have been very difficult and a lot of effort.

Instead going for the full simulation approach, the team started out creating the simplest possible tests (using ***scenarios***) and moving on to complicated ones. First challenges here were to get the system running in a testing environment. Already this required restructuring proved to make the components more decoupled. Since the feedback system uses UDP packets to get input values and send corrections, this was the obvious point to inject test data. By going small steps, the team learned on the way how the system worked and finally found ways to probe the system with well defined scenarios (e.g. orbit outliers, constant orbits ...) without ever closing the (simulation) loop.

In the meantime, there exist about 50 well defined tests run on a ***CI server***, which form a basic safety net. They are all formulated in a concise DSL (Domain Specific Language), to keep them short and clear [3]. Work still has to continue in the future to evolve the framework together with new features of the feedback system. ***Unit tests*** were not existent for the original system. However, for new features which are added, such unit test are developed and will soon also run within continuous integration.

ISBN 978-3-95450-177-9



Copyright © 2016 CC-BY-3.0 and by the respective authors

A similar approach is planned to be taken for other beam instrumentation systems in the near future (Beam Loss Monitors, Wirescanners).

### *More examples*

In the above sections we picked only some examples from projects from recent history to demonstrate the basic concepts. However, there are many other projects which follow already similar approaches. Some more worth mentioning are:

- To measure the chromaticity of the LHC, a simple application was developed in 2015 [4]. This application modulates the RF frequency measures, the tune and deducts the chromaticity values from harmonic fits to both, tune signals and RF signal. Next to standard unit tests, this project uses a similar approach than the Luminosity Server: When in ***development mode***, it has a (very simple) model running which calculates tune from the modulation and thus provides self-consistent input signals for the application.
- A special system to survey power converter currents and trigger interlocks in case of anomalies was developed for the LHC [5]. Also this system has a dedicated ***development mode*** to allow to work without any hardware access.
- YASP (Yet Another Steering Program), the standard tool for orbit steering in all CERN accelerators, also provides a ***simulation*** mode for development and debugging.

While all these examples make heavy use of their simulation modes, an integrated simulation mode across several systems is currently not implemented anywhere in the current accelerator control system at CERN. However, to achieve more ***integration tests with other components***, such a mode would be highly desirable and future efforts will have to go in that direction.

## CONCLUSIONS

Testing distributed systems, like accelerator systems, is hard but not impossible. While common practices in the software domain in general, it is still less common in our environment, especially the closer it gets to hardware (where it also is more difficult). In the previous sections we categorized the different level of tests and gave examples how they can be and how they are currently applied on different projects. In order to enable fearless development and testing of accelerator components, we consider the following approaches as important (most important first, most complex last):

1. Any kind of ***development mode*** is practically a must. This allows to develop, debug and test the component (software, hardware) in isolation and removes the risk to accidentally access e.g. a real device during development. This already enables (at least manual) ***single component integration tests***.

2. For proper operation it is crucial to have proper ***error reporting*** on all layers (Faults, Exceptions). Warnings should only be issued in case the user is able to act on them.

3. To enable post-mortem diagnosis, centrally stored and easily searchable ***tracing information*** is indispensable.

4. For any software component, ***unit tests*** are usually a standard approach. Next to providing a first safety net against regressions, they also enforce a cleaner code structure, if applied from the beginning.

5. If possible, ***automated testing*** is preferable compared to a manual approach. If tests are automated, then the step to ***continuous integration*** is a small one and is highly recommended.

6. For hardware components, periodic ***sanity checks*** during operation can ensure the proper functioning of devices and help spotting degradation early.

7. It can get relatively complex to implement ***integration tests with other components***. However, they pay off at some stage, because the only alternative for an accelerator is to stop operation and test the components interactions in production.

## REFERENCES

[1] J. Emery et al., "First experiences with the LHC BLM sanity checks", Topical Workshop on Electronics for Particle Physics 2010, Aachen, Germany.

[2] S. Van Der Meer, ISR-PO/68-31, KEK68-64

[3] S. Jackson et al., "Testing Framework for the LHC Beam-Based Feedback System", this conference.

[4] K. Fuchsberger, G.H. Hemelsoet, "LHC Online Chromaticity Measurement - Experience After One Year of Operation", this conference.

[5] K. Fuchsberger et al., "LHC Orbit Correction Reproducibility and Related Machine Protection", Proc. IPAC 2012, New Orleans, Louisiana, USA.

Copyright © 2016 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-177-9