Proceedings of the CERN-BINP Workshop for Young Scientists in e^+e^- Colliders, Geneva, Switzerland, 22 – 25 August 2016, edited by V. Brancolini ans L. Linssen, CERN Proceedings, Vol. 1/2017, CERN-Proceedings-2017-001 (CERN, Geneva, 2017)

iLCDirac and Continuous Integration: Automated Testing for Distributed Computing

J. Ebbing
CERN, Geneva, Switzerland
Karlsruhe Institute of Technology, Karlsruhe, Germany

Abstract
Detector optimization studies for future high-energy physics experiments require the simulation and reconstruction of many physics processes and detector geometries. As an efficient way of accessing the necessary computational and storage resources, DIRAC has been developed and extended to form iLCDirac, which is specialized for the applications used in the context of linear collider detector studies. We give a short introduction of grid computing and the concept of high-throughput computing behind DIRAC before explaining the unique features of DIRAC and iLCDirac. With this preparation we explain how we leveraged continuous integration to provide smooth day-to-day operations and ensure that changes to the underlying code base do not cause an interruption of the service.

Keywords
Continuous integration; iLCDirac; grid computing; quality assurance.

1 Introduction

To design experiments for future e^+ e^- linear colliders that would take the place of the Large Hadron Collider (LHC), extensive studies on different variations are performed. These Monte Carlo simulations require vast computational resources that are provided by grid computing, specifically by the Worldwide LHC Computing Grid [1], as well as the Open Science Grid [2, 3]. To use these resources as efficiently as possible, the LHCb experiment developed the grid middleware DIRAC [4], which handles software installation, job scheduling, workload management, data bookkeeping, and replica management. The resources of the International Linear Collider (ILC) and Compact Linear Collider (CLIC) communities are bundled and shared via the ILC virtual organization. The iLCDirac extension [5], which was developed on top of DIRAC, offers workflow modules for the linear collider software. The members of the virtual organization share the software for event generation, detector simulation, event reconstruction, and data analysis, as well as their computational resources.

Thus, iLCDirac has been developed to offer a simple interface for users and has been adopted generally by the linear collider community. Originally developed for the CLIC Conceptual Design Report [6], iLCDirac has been used successfully for data production and analysis for the Silicon Detector concept in the ILC Detailed Baseline Design document [5, 7] and is now also in use for the International Large Detector concept. The available resources are mostly opportunistic, i.e., not guaranteed to be available at all times; for example, there can be another user with higher priority but when the resource has been idle for some time the resource is allocated to the grid. At most, around 20,000 concurrent jobs are handled by the system [8].

2 Grid computing and high-throughput computing

2.1 Introduction to grid computing

Grid computing refers to a special kind of distributed computing, with the machines usually being heterogeneous (using hardware with differing architectures, different operating systems, or preinstalled

https://doi.org/10.23727/CERN-Proceedings-2017-001.103

libraries), geographically dispersed (worldwide) and connected to reach a common goal. It is used to provide the LHC experiments with the enormous amounts of computational resources and storage capacities necessary for functioning. The ideas of the grid were pioneered in the 1990s, mainly by Foster *et al.* [9]. In 2002, Foster released a three-point checklist [10] to determine whether a system can be called a grid. It states:

> I suggest that the essence of the definitions [...] can be captured in a simple checklist, according to which a grid is a system that:
>
> 1. co-ordinates resources that are not subject to centralized control [...];
> 2. using standard, open, general-purpose protocols and interfaces [...];
> 3. to deliver nontrivial qualities of service.

Unlike conventional supercomputers, machines of a single grid can be spread across the world. This means that, for example, the round-trip time will be rather high and the data transmission rate not as high as it would be if the machines were in the same building. This is offset by several advantages of distributed systems, e.g., not containing a single point of failure. In the case of simulations for high-energy physics it is also noteworthy that the problems are of such a kind that two machines do not have to communicate in order to finish their computation, so-called *embarrassing parallelism*. This is also referred to as a *loose coupling* of parallel tasks, which can be seen as a characteristic of grid computing (again as a result of the long-range connection). There are also designated (including undersea) cables deployed for CERN and the LHC experiments, further alleviating the issues.

2.2 The high-throughput computing paradigm

Most physical supercomputers follow the high-performance computing paradigm, aiming to maximize, e.g., the number of floating point operations per second (FLOPS) that the machine can perform. However, scientific computation tasks, such as those that arise in high-energy physics, can take much longer, making the number of floating point operations per month or year (FLOPM or FLOPY) a much more realistic and thus important metric. Owing to general downtime, idleness, and inefficiencies of the system, and many other reasons, the number of FLOPS cannot simply be scaled up to FLOPM or FLOPY. For this reason, high-throughput computing was developed, focusing on the reliability and robustness of a system that executes a large number of jobs in parallel.

To maximize the computational efficiency with the allotted resources, DIRAC follows the high-throughput computing paradigm and introduced novel concepts such as *pilot jobs*, which are described in Section 3.1, to grid computing.

3 DIRAC and iLCDirac

This section discusses the grid software developed by the LHCb experiment, DIRAC, as well as the extension for the ILC community, iLCDirac. DIRAC was originally developed to fulfil the grid needs of LHCb. However, to encourage reuse of the general-purpose DIRAC code for other applications, the LHCb-specific code was moved to its own extension, LHCbDirac. Today, DIRAC is also used by many researchers outside of collider-based high-energy physics, for example in using the Cherenkov Telescope Array [11] and the Fermi Large Area Telescope [12].

3.1 DIRAC grid middleware

DIRAC was developed to provide common services exclusive to LHCb. A special focus was put on the system requiring as little manpower for site management as possible and on the need to scale to forty or more sites easily. A *pull* model has been implemented for scheduling so that instead of a centralized job

queue looking for resources for its jobs, each free resource site will ask a central service for a free job to execute. This has benefits for scalability and efficiency in the high-performance computing paradigm. An innovation introduced to grid computing by DIRAC is the introduction of so-called *pilot jobs*, which are first sent to a free resource. As opposed to actual jobs, these pilot jobs do not require large input files, making them a lot cheaper to send; their purpose is to check whether a following normal job can be executed in this environment. If this check fails, a real job is not sent and much time is saved, compared with the case in which the normal job is sent immediately. This extra step before an actual job submission is an improvement, since grid resources can be unreliable. Without pilot jobs, a defective site would continuously request jobs, putting unnecessary strain on the system and not doing anything productive. Since the resources of the grid are, by nature, heterogeneous, the purpose of DIRAC is to provide homogeneous access to these heterogeneous resources; this is achieved through the job interface—a user is usually not concerned with selecting sites on which the jobs are run.

3.2 Extending DIRAC for the linear collider community

To use DIRAC in the linear collider community, the iLCDirac extension has been developed. Analogous to LHCbDirac for LHCb specifics, this system encapsulates the software specifics for the needs of the linear collider community. It builds on DIRAC by using the core, configuration, and general framework of DIRAC, as well as the file catalogue, workload management for job handling, workflow management for job submission, and existing transformation system. Owing to the linear-collider-specific software necessary for grid jobs, the software installation in iLCDirac changed to downloading tarballs from the grid, which was later extended to optional use of the shared area for software installations. Interfaces for the aforementioned specific applications were added, to simplify job submission for the users, as well as some internal modules for typical ILC use cases.

4 Software testing

Since all sufficiently complex software contains bugs the need for quality assurance in software development arises. This is especially true for the large-scale programs being developed today, as the hardware and maintenance costs of the necessary data centres are high. Testing has been researched extensively in the past, with, e.g., Dijkstra stating “Testing shows the presence, not the absence of bugs.” [13]. Owing to the complexity of distributed computing and the importance of providing a working system to the physicists working with DIRAC/iLCDirac, extensive testing is paramount.

There are many ways to categorize software testing. One of the most important is automated testing, by contrast with manual testing; automated testing refers to small test programs that test a larger piece of software.

Another important distinction is the knowledge of the underlying system: in whitebox testing, the test knows how the system under test works internally and can use this information, whereas a blackbox test only knows the public application program interface (API) of the software and thus provides an input and checks the final output of the program. Since all types of testing generally have advantages and disadvantages, it is usually advised to mix different categories in real software projects with the concrete distribution, depending on the specific software under test.

4.1 Test hierarchy

Simultaneously with the proposal of agile programming methods, such as extreme programming, test automation became an important topic in software development and is a core part of many agile methods. In short, agile software development is based on many small iterations, which means that a small part of the system is planned in close collaboration with the customer and implemented as well as deployed in a short timeframe. This flexibility contrasts with the long planning phases used previously, which often did not adequately reflect customers’ needs. Owing to this workflow, core parts of the software are changed

multiple times, requiring the entire software package to be tested again after each small change. This presents an excellent use case for automated tests, as they can be run quickly and at no cost.

Writing smaller programs that test a large piece of software automation presents several advantages.

- It is much faster than executing the same test manually (although, in some cases manual tests might in fact be *faster* than automated ones).
- When a test suite is developed, its execution for each new version guarantees that the old functionality (for which the tests check) is not broken with the update, preventing so-called *regressions*.
- It allows for new software development models, such as test-driven development, in which the tests are specified first and then the actual code is developed.
- It facilitates the development of new features, since the developer can concentrate on his or her part of the program only.
- For the same reason, it saves a lot of work when refactoring older parts of the code since the basic functionality is ensured. Without these tests, a developer tasked with refactoring faces the vicious circle that the code is too complex to be tested before refactoring, but the refactoring might change the behaviour of the code.

To structure the problem better, automated tests are usually divided into several types. *Unit tests* are the most low-level tests and should usually be the most numerous. They are specific to a 'unit' of software—in object-oriented languages, usually an object or a method—and test whether it fulfils its design contract by setting a state, providing an input, and checking for the correct output and the correct new state of the system. Dependencies on other parts of the code should not be tested and are usually avoided through special mock objects that replace the original dependencies. This process is colloquially known as mocking. A unit test should make as few assumptions about its environment as possible and run as quickly as possible (a usual aim is of the order of several milliseconds). This means that all calls to databases, file systems, network connections, etc., in the original code should be mocked.

To check for the correct interaction of code units, *integration tests* are written. Depending on the application, these tests can take much longer than unit tests. For example, an integration test might involve setting up a database with some basic values, performing actions in a graphical user interface (GUI), and checking whether the correct results are displayed. Owing to these more complex interactions between modules and, e.g., interactions of a user with a GUI, these tests can be more difficult to automate.

Finally, *system tests* test the basic functionality of the system as it would be used later, e.g., not setting up any system state beforehand. These tests are usually very complex; thus, only a few of them will be written. They can take a very long time to run and may require special infrastructure to be executed. System tests can be even harder to automate than integration tests and may also be performed manually.

The correct use of these types of test is very specific to the software project in question. For example, a custom sophisticated data structure implementation will require many unit tests with a large underlying dataset and predefined expected behaviour, whereas a simple application for a non-tech-savvy user with a GUI will require a larger proportion of system tests.

4.2 Testing metrics

Especially in commercial software development, project metrics are an important factor to measure with respect to, e.g., the progress and success of a project. However, these metrics are always just a heuristic and the specific project will determine many of the testing requirements. For example, a compiler would need as many tests as possible and every line should be touched at least several times, whereas simple end-user application code does not have such high requirements, since the code complexity is far lower. For test coverage, different metrics have been proposed, each with its own shortcomings.

The most simple metric is *statement coverage*. This involves leaving all lines of code untouched by default, then executing the test suite and marking every line that is being executed as touched. At the end of the test, the coverage is the number of touched lines or statements divided by the number of total lines. This is a very weak metric and should not be used, for many reasons. It is easily possible for code still to contain lots of bugs even though the statement coverage is 100%.

A slight improvement is *branch coverage*, which is concerned with the amount of branches that the code contains. An if-condition has two branches, the if-part and the else-part. Similarly, loops can be executed at least once or never. Then the coverage is calculated as the number of executed branches divided by the number of total branches. This metric should be seen as the bare minimum; more complex metrics can lead to an improvement in software quality. To give an intuition of how defects can still happen with this metric, consider a method that consists of two if-conditions. The bug occurs when the first if evaluates to `true` and the second if to `false` but the two test cases that have both ifs evaluate to true and false, respectively, achieve 100% branch coverage.

A theoretical metric is *path coverage*, which considers all possible paths through a program. This could lead to provable correctness but the number of tests required for this metric leads to a combinatorial explosion even in very simple programs. Writing a test suite with 100% coverage could be replaced by simply writing a table that maps all possible input values to the correct output values, defeating the purpose of the software in the first place.

4.3 Continuous integration

Continuous integration is a concept that is critical for most agile methods. It revolves around writing a large base of unit tests that can be executed very quickly and then setting up an automatic system that executes these tests whenever a developer commits a change to the code base. This prevents regressions by adding a unit test for every bug that has been found and fixed. If a new change breaks old behaviour, the corresponding test will fail and the code base can be fixed very quickly, since the developers get almost immediate feedback. Aside from the benefits this has for larger collaborations due to a healthy code base and preventing repeating the same work again, this technique also allows for fast release cycles.

5 Continuous integration in iLCDirac

The iLCDirac team uses continuous integration to ensure that bugs do not reach production. Since the project uses GitLab, this is implemented using GitLab-CI. Several dedicated test machines run the suites on each commit in parallel, resulting in short feedback times (around 10–15 min for all suites, including system tests). While executing the tests, we also measure and report branch coverage and use tools to analyse overall code quality. The tests are run on both CERN CentOS 7 and Scientific Linux CERN 6 (as both operating systems are in use) using docker images. This has the additional advantage of being able to use test specific environments.

Several test suites have been written.

1. Code style checks and checks for basic errors with Pylint. Since Python is a dynamic and interpreted language, this is necessary to display some basic errors that a compiler would catch. A common code convention is necessary for any software project.
2. A large suite of unit tests, which is currently being extended. These are written on a per-class basis and test individual methods, using blackbox testing for public API methods and whitebox testing for private methods.
3. System tests, mainly sample job submissions for the simulation software used by the linear collider community. A collection of configuration and input files for tests is used and success of the job is expected.
4. System tests for the storage elements used on the grid. These upload a file with random bits to

storage elements, replicate it to and delete it on several storage elements, and retrieve the copies and the original file. Consistency between the original and the downloaded file is expected and all operations are checked for success.

Where necessary, a script performs a full installation of DIRAC and iLCDirac in a docker image before executing these suites. To perform manual testing as well, newest versions are not automatically pushed to production. These updates are handled manually once a certain maturity of a version has been proved. During my technical studentship, branch coverage has improved from 32.2% to 52.1%.

6 Summary

To conclude, we established an automated testing system for iLCDirac that leverages several types of automated tests to provide high software quality and prevent regressions. Since it is automatically executed on each commit to the main code base, we ensure a healthy code base and are able to pinpoint new bugs to concrete commits very quickly. As anecdotal evidence, we recently updated to a new DIRAC version without any major problems. The usage of continuous integration revealed bugs in the original DIRAC that could be fixed before causing problems in production. Continuous integration also proved useful for refactoring code, which is a large part of everyday work. The current development focus in the continuous integration area is the further extension of the unit test suite.

References

[1] C. Eck *et al.*, LHC computing grid: technical design report, CERN-LHCC-2005-024, (CERN, Geneva, 2005).

[2] R. Pordes *et al.*, *J. Phys. Conf. Ser.* **78** (2007) 012057. http://dx.doi.org/10.1088/1742-6596/78/1/012057

[3] I. Sfiligoi *et al.*, The pilot way to grid resources using glideinWMS. 2009 WRI World Congress on Computer Science and Information Engineering, 2009, vol. 2, p. 428. http://dx.doi.org/10.1109/CSIE.2009.950

[4] N. Brook *et al.*, DIRAC-distributed infrastructure with remote agent control, Computing in High Energy and Nuclear Physics (CHEP03), La Jolla, CA, 2003, cs.DC/0306060.

[5] C. Grefe *et al.*, *J. Phys. Conf. Series* **513** (2014) 3.

[6] L. Linssen *et al.* (eds.), Physics and detectors at CLIC: CLIC conceptual design report (2012).

[7] T. Behnke *et al.*, The International Linear Collider technical design report—volume 1: executive summary (2013).

[8] A. Sailer. Talk: iLCDirac and the grid, ECFA-Linear Collider Workshop, 2016.

[9] I. Foster and C. Kesselman (eds.), *The Grid: Blueprint for a New Computing Infrastructure* (Morgan Kaufmann Publishers Inc., San Francisco, CA, 1998).

[10] http://www.mcs.anl.gov/~itf/Articles/WhatIsTheGrid.pdf, last accessed September 26th 2016.

[11] L. Arrabito *et al.*, *J. Phys. Conf. Series* **664** (2015) 032001.

[12] L. Arrabito *et al.*, *J. Phys. Conf. Series* **513** (2014) 032003.

[13] J.N. Buxton and B. Randell (eds.), Software Engineering Techniques, Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969 (NATO, Scientific Affairs Division, Brussels, 1970).