ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV035

CONTINUOUS INTEGRATION FOR PLC-BASED CONTROL SYSTEM DEVELOPMENT

B. Schofield*, J. Borrego, E. Blanco, CERN, Geneva, Switzerland

Abstract

Continuous Integration/Continuous Deployment (CI/CD) is a software engineering methodology which emphasises frequent, small changes committed to a version control system, which are verified by a suite of automatic tests, and which may be deployed to different environments. While CI/CD is well established in software engineering, it is not yet widely used in the development of industrial controls systems. However, the advantages of using CI/CD for such systems are clear. In this paper we describe a complete CI/CD pipeline able to automatically build Siemens `Simatic` Programmable Logic Controller (PLC) projects from source files, download the program to a PLC, and run a sequence of tests which interact with the PLC via both a Simulation Unit Profibus simulator and an OPC Unified Architecture (OPC UA) interface provided by `Simatic NET`. To achieve this, a Google Remote Procedure Call (gRPC) service wrapping the `Simatic` Application Programming Interface (API) was used to provide an interface for interacting with the PLC project from the pipeline. In addition, a Python wrapper was created for the Simulation Unit API, as well as for the OPC UA interface, which allowed the test suite to be implemented in Python. A particle accelerator interlock system based on Siemens S7-300 PLCs has been taken as a use case to demonstrate the concept.

INTRODUCTION

In software engineering, Continuous Integration (CI) refers to a method of development in which changes are regularly incorporated into a central repository. It is very often associated with the presence of build and test automation, in which code is automatically compiled, and a set of tests run. The objective of the methodology is to provide early detection of bugs introduced by code changes, and to simplify workflows in the case where there are several developers working on a single code base (the alternative is to perform periodic merges of the individual developers' branches, which may be very complex if many changes have been made).

Continuous Deployment (CD) is perhaps less well defined, and entails at least the automatic release of some artefact of the automated build process in the CI stage. It may also involve the fully automatic deployment of the artefact in a production environment.

Software for PLC-based control systems traditionally does not follow CI/CD principles, for a number of reasons. Generally, proprietary engineering tools are used as the development environment, in which code is written, compiled and downloaded to the PLC.

In general there is no support for external version control systems to be used for the source code, at least as far as CERN's standard PLC suppliers are concerned (Siemens, Schneider). Instead, often the full project is included in version control as the collection of files and data used by the engineering tools, provided one is using version control at all. Tracking code changes is difficult, and merging branches is effectively impossible with such a workflow. Automation of the building of PLC projects is not trivial, as not all engineering tools provide easy access from scripting languages to these functionalities. Finally, automated testing proves to be cumbersome for a number of reasons that will be elaborated in later sections.

The question addressed in this article is whether it is technically feasible to implement a CI/CD workflow for PLC-based controls development, and if so, whether such a workflow is practicable and useful in a real-world application.

In order to address the first point, set of tools will be introduced which aim to overcome the obstacles preventing the adoption of CI/CD for PLC-based control system development. Tools for automation of the build process are proposed, with the focus on Siemens `Simatic` applications, although tooling for other engineering tools has also been developed. An approach for implementing automated testing of the complete PLC program is described, consisting of an interface to a fieldbus simulator, as well as an OPC UA interface to the PLC and Supervisory Control And Data Acquisition (SCADA) layers of the control system.

To demonstrate these tools, and illustrate their potential, a use case consisting of an interlock system for the Large Hadron Collider (LHC) is presented. Major updates and refactoring of this control system have been enabled by employing the CI/CD workflow presented in this article.

The article begins with addressing the question of the tooling required to automate the building of PLC projects. After that, automatic testing is addressed. Finally, the use case is explained and details of the proposed workflow are given within the context of that project.

TOOLS FOR AUTOMATING PLC PROJECT BUILDING

Intended Workflow

In order to implement CI/CD for PLC-based applications, it is necessary to adopt a similar underlying development workflow to that used elsewhere in software engineering. Fundamentally, this means adopting the use of version control for all source code. The source code is then compiled to produce some form of output, for example executable programs or libraries, for one or more target systems. These outputs in themselves do not need to be included in version

* Corresponding Author. E-mail: brad.schofield@cern.ch

control, as they can simply be recreated from a specific state of the versioned source code. The outputs are often referred to as *artefacts*. The CI/CD approach is to automate this process of creation (and subsequent deployment) of artefacts.

In the majority of mainstream programming languages, the source code simply consists of text files. The most widely used version control systems are therefore also based on the use of text files. If it is intended for the complete source of a PLC program to be stored in a version control system, this introduces some restrictions. Of the IEC-61131-3 languages, only the languages which can be represented in text, namely Structured Text (ST) and Instruction List (IL) are suited to this approach. However, the requirement to use non text-based PLC languages does not preclude the use of the proposed approach; it may be the case that a part of the code base is implemented in text-based languages, and other parts in non text-based languages, of which the former are tracked in an external version control tool and the latter kept in the PLC engineering tool.

PLC Engineering Tools

As previously mentioned, Programmable Logic Controller (PLC) engineering must be performed using the proprietary engineering tools supplied by PLC vendors. The interface for such tools is typically a Graphical User Interface (GUI), in which importation of code from external sources, code editing, compilation, configuration of the target, and downloading of the program are performed. In general, these tools do not have good support for version control systems, meaning that it is difficult to maintain consistency between source code that is maintained in version control, and the resulting compiled version of a project. This is in essence the core problem being treated in this paper. While it is possible to use dedicated version control tools for PLC projects such as `versiondog` [1], they lack some of the more advanced features of version control systems employed in standard software development like `git`. They also implicitly track all changes to the compiled project, whereas the desired workflow outlined here aims at keeping only the text-based sources in version control, excluding any derived binaries and other non-essential data files. In order to achieve the desired workflow, it is necessary to automate tasks ranging from importation of external sources, to compilation and ultimately downloading of the project to a PLC. If this can be achieved, then the CI/CD workflow will be able to guarantee consistency between source files in the repository and a program running on a PLC, something which is currently not possible with existing tools.

In this work we focus on the engineering tools for Siemens S7-300 PLCs, since these are present in the system at hand. These PLC projects are developed using `Simatic Step 7`, which exposes a C#/Visual Basic API. We developed a C# library which allows us to import source files, build the project and download it to either a test or production PLC. We then designed a simple gRPC service, allowing our C# server to fulfil requests and interact directly with the Step7 library, which can be launched from remote clients, implemented in whichever language is convenient. We use a Python client due to its simplicity of use and to match the environment for running the integration test suite. Figure 1 shows the client/server architecture for the proposed S7 gRPC service.

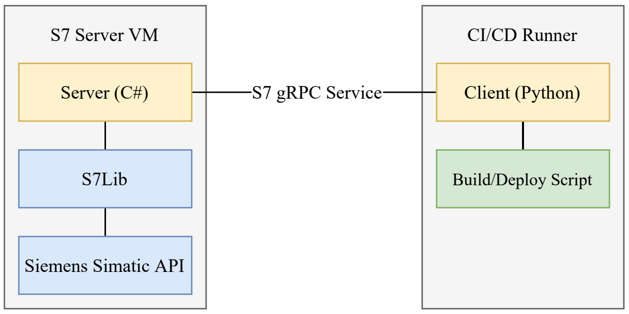

Figure 1: Architecture of the `Simatic` build tools.

The S7 gRPC service exposes several methods, namely:

- `CreateProject` to create an empty `Simatic` project given the project name and destination path;
- `ImportSource` to import a source file into project;
- `ImportSymbols` to import a symbol table file into project;
- `CompileSources` to build target sources into blocks;
- `CompileAllStations` to build the hardware configuration for each station in the `Simatic` project;
- `StartProgram` to start/restart a specific program on the target CPU;
- `StopsProgram` to stop a specific program on the target CPU;
- `DownloadProgramBlocks` to download a specific program to target CPU.

Finally, we have also developed similar tooling to support newer Siemens S7-1500 and Schneider CPUs, although these are not in the scope of this article.

AUTOMATED TESTING FOR PLC PROGRAMS

With automated build and deployment tools in place, the last remaining piece required to implement a complete CI/CD workflow is the automated test suite. Automated testing of PLC programs is challenging for several reasons. For the majority of modern programming languages, there are numerous unit testing frameworks available, which make the implementation of unit tests almost trivially simple. In addition, it is generally not difficult to create a build configuration which will create test executables which can be run locally, for example on the developer's machine. This is

[1] `https://auvesy.com/en/versiondog`, as of October 21, 2021

however not the case for PLC programming languages. In order to implement true unit tests for PLC code, frameworks specific to a PLC type would have to be implemented. This topic is explored in [1], in which a unit testing framework for Siemens PLCs is developed. For integration testing, the problem is further complicated by the difficulty of 'mocking' the necessary interfaces (for example fieldbuses and supervision systems). Regarding the target on which the test code shall be run, in the case of PLCs this must be either a physical PLC CPU, or a simulator. While the simulator approach may be attractive, currently not all PLC simulators support the functionality required to implement complete test suites.

The approach to PLC program testing proposed here is to generate a PLC program which is either identical, or very close to the final program required in production. This program is then deployed to a physical PLC. The test suite is then implemented externally, and utilises a number of interfaces to interact with the PLC in order to run the tests. The interfaces to the PLC can be summarised as acting on three levels, namely the input-output or fieldbus level, the PLC program level, and the supervision level. These interfaces will be described in detail in the next sections, before describing how they can be combined into a single test suite.

In software development there's a clear distinction between unit testing, integration testing and system testing. Unit tests focus on verifying the correctness of individual functions or modules in isolation. Integration tests aim at ensuring the module's interface is correct and it behaves as expected when interacting with other modules. Finally, system testing verifies that a completely integrated system fulfils its requirements. Our proposal cannot be said to consist of unit tests, at least as far as the PLC source code is concerned. Instead we introduce a set of tests which verify that the control system fulfils its task accordingly, which is more strongly aligned with system testing. Even though we use several communication interfaces and thereby implicitly test them, they are not the focus of the test, so they cannot be considered integration tests. Nevertheless, we still mock the PLC inputs and evaluate outputs at both the PLC and SCADA layers.

Fieldbus Simulators

A PLC application will almost invariably need to interact with the physical world via sensors and actuators. The input-output (IO) hardware required to do this may be installed locally to the PLC CPU, but is more generally installed in a distributed manner, and connected with the PLC via a fieldbus. The fact that this interface is standardised can be exploited for testing purposes, if simulators for such fieldbuses are available. In this article, the focus is on Siemens PLCs, and therefore Profibus and Profinet fieldbus simulators will be taken as examples. Siemens provides hardware which allows the simulation of arbitrary hardware configurations for both Profinet and Profibus. The units are controlled via an Ethernet interface, and a C API is available. For the purposes of automatic testing, such a simulator can be connected to a test PLC, and a CI/CD pipeline could be used to configure the simulator (by loading a specific hardware configuration), and control IO values, both via the API. In order to more easily access the functionality of the simulation unit from the automated build and test suite, a Python wrapper `py-simulation-unit` was created.

PLC Interaction via OPC UA

While the fieldbus simulator provides a way to mock field level input to the PLC, from the point of view of writing tests it may be valuable to access the internal state of the PLC program. This is fairly straightforward to achieve provided the target PLC can expose tagged variables in an OPC UA server. In the case of Siemens PLC, Siemens `Simatic NET` software provides such an OPC UA server, which runs in a separate workstation. Newer generations of PLCs also have onboard OPC UA servers, further simplifying the setup.

In [2], OPC UA was used as the exclusive interface for testing PLC applications. This was possible because the applications under test were all created with the UNICOS framework [3], the structure of which provides a simple way to 'override' the hardware IO interface with values written over OPC UA. This effectively means you can mock the IO at the PLC program level. Depending on the way in which an application is written, it may not always be possible to do this, and IO simulation may be necessary.

In this work, the OPC UA client API from the open source LGPL Pure Python OPC UA[2] package is used.

Supervision Systems

A PLC-based control system most often interacts with a SCADA system, in which operators can observe the status of the system, and send commands to the PLC. In certain cases there may be additional logic implemented in the SCADA system, which provide further inputs to the PLC. Rather than mocking the SCADA interface (which in the use case presented in this article consists of Siemens S7 communication with a CERN-specific protocol built on it), it was decided to instantiate a SCADA system specifically for the test setup, and to interact with it via OPC Unified Architecture (OPC UA). This interaction allows the test suite to 'force' operator actions and logic outputs to the PLC, and thus provides a very realistic test setup. An added benefit is that the test suite can be used for manual validation of the graphical features of the SCADA system, as shown in Fig. 2. Effectively, by visual inspection of the SCADA faceplate it is possible to monitor the automated test execution in the lab setup. The SCADA system used here is built with WinCC OA, using features from CERN's UNICOS framework [3].

[2] `https://github.com/FreeOpcUa/python-opcua`, as of October 21, 2021

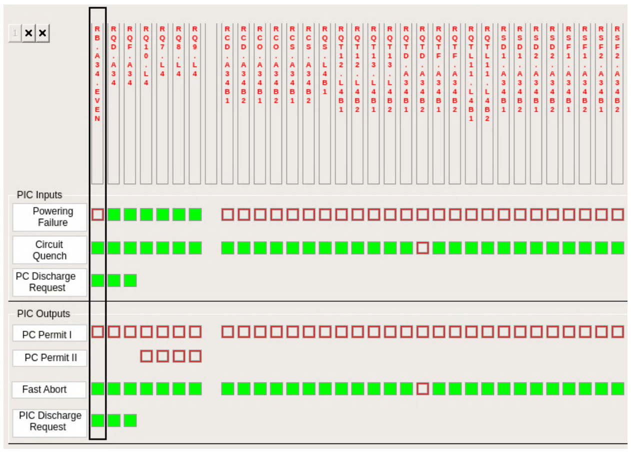

Figure 2: Part of the SCADA faceplate for one Powering Interlock Controller PLC. The automated test suite can be used to drive inputs to manually validate the supervision layer.

USE CASE: LHC POWERING INTERLOCK CONTROLLER

An example of the utilisation of the proposed CI/CD workflow can be found in the Powering Interlock Controller (PIC) of the LHC. This system controls the powering of the superconducting magnets of the LHC, and consists of 36 Siemens S7-319 PLCs, physically distributed around the 27 km circumference of the LHC as shown in Fig. 3. Each PLC is responsible for managing the interlocks for a certain number of electrical circuits powering the magnets. Depending on the types of magnets being controlled, the PIC is required to monitor different types of signals, and perform different interlock actions. All 36 PLCs utilize a common generic code base, which is parameterized in each PLC using source files which determine input-output mappings and circuit types [4].

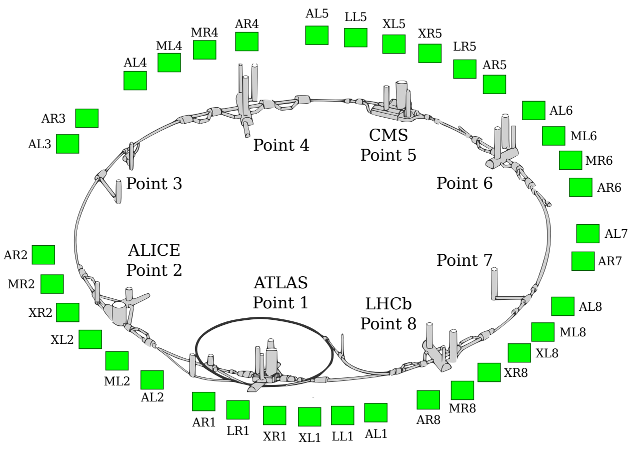

Figure 3: Layout of the LHC showing PIC PLC locations. Each PLC is shown as a green box with the corresponding name.

Legacy Workflow

Prior to implementing a CI/CD workflow, the method used to update the generic code was to edit it in each `Simatic` project. This was time-consuming and error prone. Deployed versions of the PLC projects were kept in `versiondog`, which enabled tracking of changes to each individual `Simatic` project. Testing was performed manually with a hardware-based lab setup. Consistency between the code depoloyed on the lab setup, and that deployed in the production system had to be handled by the developer.

CI/CD Workflow

The starting point of the CI/CD workflow was the creation of a `GitLab` repository containing a set of sources (ST and IL files) for both the generic code (the same for all PLCs) and the configuration code (different files for each PLC). A set of 'base' `Simatic` projects, containing the hardware configurations of each PLC, are also included in the repository. It should be noted that these projects are only used as inputs to the workflow; that is, the resulting complete `Simatic` projects are not versioned in `git`.

Lab testing workflow The automated testing is performed in the lab, using an identical PLC to those used in production, connected to a Siemens Profibus Simulation Unit. An additional `Simatic` project containing the configuration of this PLC is present in the repository; the hardware configuration of which is the basis of the Simulation Unit configuration.

The build stage of the CI/CD workflow uses a Python build script to import the generic sources as well as the specific configuration sources of the configuration to be tested. The resulting program is compiled. Finally, the complete PLC program is downloaded to the lab PLC. In order to update the OPC UA server used for the testing, the script also downloads to the OPC UA station.

With the build stage complete, the test stage can begin. This simply involves calling `pytest` on the complete test suite. Test fixtures manage the connections to the Simulation Unit, as well as the OPC UA servers for both the PLC and the SCADA. An example of a `pytest` fixture is as follows (for the PLC client):

```
@pytest.fixture
def plc_client() -> Client:
    """Returns an OPC UA client connected to the PLC"""
    client = Client(CLIENT_PLC_URL)
    client.connect()
    yield client
    client.disconnect()
```

Listing 1: Fixture for OPC UA PLC Client

The SCADA client fixture follows the same pattern. The simulation unit fixture (called `simba` in this use case) is slightly more complex as the hardware configuration is loaded.

A specific test may use any combination of these fixtures, depending on what needs to be accessed in the test. The following is a very simple test to verify the magnet quench detection status for the main dipole magnet circuit:

```
def test_quench_abort_status_rb(plc_client, simba):
    """Main Dipole Quench Status test"""
    plc = plc_client.get_root()
    status_abort = plc.get_child("7:A_A_1_ST_ABORT")
    quench_status =
     ↪  plc.get_child(['7:1_Instance1_Circuit1',
     ↪  '7:Quench_Status'])

    simba.write_io_bin('S000I4.1', 1)
    wait()
    assert status_abort.get_value() is True
    assert quench_status.get_value() is False

    simba.write_io_bin('S000I4.1', 0)
    wait()
    assert status_abort.get_value() is False
    assert quench_status.get_value() is True
```

Listing 2: Example Test for the Power Interlock Controller System

It can be seen that the inputs are manipulated using their hardware addresses directly, via the `simba` fixture. If desired, an abstraction layer based on the signal list of a specific project could be implemented to simplify the reading and writing of the tests. The test assertion is based on accessing the instance data block of a specific function block representing a state machine corresponding to the main dipole circuit, and verifying that a particular property is correctly set. This illustrates the usefulness of the OPC UA connection to the PLC, as this information may not necessarily be accessible via other interfaces such as the SCADA.

Naturally, the more complete functionality tests are more complex than the examples reproduced here, but all tests follow the same principles. All three interfaces are used to drive the program to the desired initial state, and then test inputs can be given, representing either field or operator input. The interface to the internal PLC data allows many ‘fine-grained’ assertions to be made, such that if a test fails, it is easy to identify the location in the code that caused the failure.

LHC build workflow With the tests passing, it is now possible to build all 36 PLC projects ready for deployment to the LHC. For this build stage, the build script is parameterized by the different subsectors of the LHC, allowing some control over which PLC projects are built. This can be useful as interventions are typically performed on one sector of the LHC at a time, and the complete build pipeline for all 36 projects takes approximately 2 hours to run. Currently, automatic deployment is not performed for the production PLCs although there is no technical barrier for this. Instead, the artefacts of the build pipeline (i.e. the `Simatic` projects) are added as a new version to `versiondog` and downloaded manually. The complete conceptual workflow is illustrated in Fig. 4.

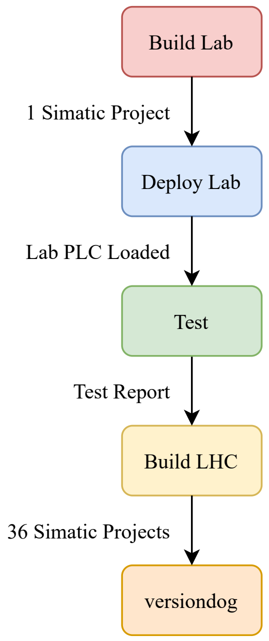

Figure 4: The conceptual CI/CD workflow. Labels by the arrows describe the artefacts produced by the preceding stage.

Tangible benefits of the CI/CD workflow In terms of the time saved for simply deploying a code change, the current workflow (with manual download) is estimated to reduce workload by approximately an order of magnitude. Manually updating each project would take at least a day, whereas simply downloading the resultant artefacts can be done in approximately one hour. With automated deployment, no developer input would be required after committing the changes to the repository and triggering the pipeline.

Naturally, the development time saved by having access to a full automated test suite is not as easily estimated, and must also be offset by the time taken to develop and maintain the test suite itself as well as the Continuous Integration/Continuous Deployment (CI/CD) infrastructure. Nevertheless, in the scope of this use case it has been clear that the CI/CD workflow has greatly facilitated development of new features, the refactoring of the code, as well as identification and resolution of bugs.

CONCLUSION

In this article a complete CI/CD workflow for PLC application development is proposed, along with a set of tools which enable all of the necessary stages of such a workflow. In summary, these tools are:

1. A gRPC service enabling scripting of many operations needed during PLC project engineering, including importation of sources, compilation, and download;

2. A Python wrapper for the Siemens Simulation Unit API, allowing configuration and manipulation of simulated IO from Python;

3. An example of an automated test suite, implemented in Python and taking advantage of existing testing packages (`pytest`) as well as an open source OPC UA package to interface with the PLC and SCADA.

The complete CI/CD workflow has been illustrated using the Powering Interlock Control system of the LHC.

Future Work

Currently, our test suite still refers to each simulated Profibus I/O channel by a tag name defined in the Simulation Unit project. A simple yet effective improvement would be to create an additional shallow abstraction layer that maps a meaningful device field to each I/O tag name. This would considerably improve the test readability.

Additionally, it seems feasible to automatically upload the PLC project built by the CI pipeline directly to versiondog, preserving a connection to a tagged commit in version control. This would effectively give us the best of both worlds: being able to quickly deploy to a running PLC while knowing exactly which source code is responsible for that specific version of the program.

Our current proposal still relies on manually creating base projects which include the hardware description. The need for these could be eliminated by just creating an empty project, automatically generating the hardware configuration and importing it as part of the CI/CD workflow.

Finally, one can think of how we can effectively implement unit testing at the PLC source code level by leveraging PLC simulators, namely with support for cycle-by-cycle execution, as is explored in [1].

REFERENCES

[1] G. Sallai, E. B. Viñuela, and D. Darvas, "Testing Solutions for Siemens PLCs Programs Based on PLCSIM Advanced," in *Proc. ICALEPCS'19*, (New York, NY, USA), ser. International Conference on Accelerator and Large Experimental Physics Control Systems, JACoW Publishing, Geneva, Switzerland, Aug. 2020, pp. 1107–1110, ISBN: 978-3-95450-209-7. DOI: `10.18429/JACoW-ICALEPCS2019-WEPHA018`. `https://jacow.org/icalepcs2019/papers/wepha018.pdf`

[2] B. Schofield, E. B. Viñuela, and J. Borrego, "Continuous Integration for PLC-based Control Systems," in *Proc. ICALEPCS'19*, (New York, NY, USA), ser. International Conference on Accelerator and Large Experimental Physics Control Systems, JACoW Publishing, Geneva, Switzerland, Aug. 2020, pp. 1527–1531, ISBN: 978-3-95450-209-7. DOI: `10.18429/JACoW-ICALEPCS2019-WESH4003`. `https://jacow.org/icalepcs2019/papers/wesh4003.pdf`

[3] P. Gayet, R. Barillere, *et al.*, "UNICOS a framework to build industry-like control systems, Principles and Methodology," in *Proc. 10th Int. Conf. on Accelerator and Large Experimental Control Systems (ICALEPCS'05)*, Geneva, Switzerland, 2005.

[4] M. Zerlauth, C. Dehavay, B. Puccio, R. Schmidt, and E. Veyrunes, "From the LHC reference database to the powering interlock system," in *9th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2003)*, Oct. 2003, pp. 395–397.