ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WESH4003

CONTINUOUS INTEGRATION FOR PLC-BASED CONTROL SYSTEMS

B. Schofield*, J. Borrego[1], E. Blanco Vinuela, CERN, Geneva, Switzerland
[1]also at IPFN - IST, Universidad de Lisboa, Portugal

Abstract

Continuous integration is widespread in software development, but a number of factors have thus far limited its use in Programmable Logic Controller (PLC) application development. A key requirement of continuous integration is that build and test stages must be automated. Automation of the build stage can be difficult for PLC developers, as building is typically performed with proprietary engineering tools. This has been solved by developing command line utilities which use the APIs of these tools. Another issue is that the program must be deployed to a real target (PLC) in order to test, something that is typically easier to do in other types of software development, where virtual environments may easily be used. This is solved by expanding the command line utilities to allow fully automated deployment of the PLC program. Finally, testing the PLC program presents its own challenges, as it is typically undesirable to alter the program in order to implement the tests natively in the PLC. This is avoided by using an industry standard protocol (OPC-UA) to access PLC variables for testing purposes, allowing tests to be performed on an unaltered program.

INTRODUCTION

Continuous Integration (CI) attempts to ensure the consistency of a project, by regularly and automatically integrating work from multiple developers into a single shared main version. Essentially its goal is to detect breaking changes as early as possible, by automatically running a set of tests when code changes are made. This process is commonly split into three automated stages, namely project build, deploy and test. Tests should be carefully designed, making sure the correctness of a software application is kept as new features are added.

Programmable Logic Controllers (PLCs) are the most common means of implementing industrial control systems. Such control system applications are often large and complex, and their specifications and requirements may change during their development or operation. For these reasons, it would be desirable to employ CI tools when developing these applications. However, automating the building, deployment and testing of PLC applications is a non-trivial endeavour. The main reasons for this include:

- Program compilation and deployment is typically done in proprietary engineering tools, which often do not support easy automation of these tasks;
- Having to resort to a physical PLC as a test target, as accurate and feature-complete simulators are not always available;
- The need to significantly modify the target test project, as normally it is required to add auxiliary interfaces to allow testing.

Previous attempts to implement a PLC testing environment have resorted to using the Supervisory Control And Data Acquisition (SCADA) stack to exchange data with PLCs. However, this approach is not without its problems. Firstly, many internal variables often cannot be accessed via the SCADA interface, making it difficult to interact with the PLC program. Furthermore, relying on SCADA for testing the PLC program introduces a dependency on the communication protocol between them.

Rather than using a real PLC one could turn to simulation software, which can even provide additional desirable features. For instance, SIMATIC PLCSim Advanced is a simulator for Siemens S7-1500 PLCs which supports fine-grained cycle by cycle execution and breakpoints. However, each simulator and its set of engineering tools is specific to a model or set of PLC models and thus require considerable effort to integrate into a testing suite directly.

Other methods seek to translate the native PLC code to widespread languages such as Java or C [1] which run on x86 architectures and have support for step-by-step debugging.

We propose a novel workflow that allows us to write tests for generic PLC-based systems which removes the dependence on a SCADA stack and requires minimal changes to the program under test. The tests description should be readable yet powerful and able to describe high-level behaviour concisely. To achieve this we implement a Python framework and a proof of concept test suite. The communications interface employs OPC Unified Architecture (OPC-UA) [2], which provides read and write access to any symbol-mapped variable in the PLC memory during runtime. OPC-UA is an open communication protocol for industrial automation which supports multi-platform communication stacks. The automation of the build and deployment phases of the CI pipeline for the PLC are carried out by developing tools which leverage the Application Programming Interfaces (APIs) of the engineering tools provided by PLC manufacturers.

As long as a test PLC can communicate with an OPC-UA server, we can interact with it via a client running on virtually any platform, written in whichever programming language fits the software stack employed by the testing framework. What is more, we conceive that by providing a single test description that uses such an OPC-UA client, we can run tests on any PLC – provided smaller structural differences in the interface exposed by OPC-UA are taken in consideration.

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

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WESH4003

UNICOS

At CERN, the UNified Industrial COntrol System (UNICOS) [3] framework has been developed in order to facilitate the implementation of industrial control applications which span mainly the control and supervision layers. It provides developers with a hierarchy of process automation objects, ranging from low-level I/O, to physical devices, to high-level control abstractions. UNICOS aims to abstract away from the underlying implementation required for a specific PLC model, and automate the creation of both PLC projects as well as SCADA monitoring interfaces from a given control system specification. Our work tries to extend this approach to automatic testing of the generated PLC projects at the level of the process objects defined in UNICOS, and incorporate it in a continuous integration pipeline.

UNICOS is composed of several components employed in different accelerator systems. We direct our attention to UNICOS Continuous Process Control (UCPC) [4], which provides a methodology for creating a specification file, in which the UNICOS objects are defined. From this file, we can automatically produce the instance and logic code for the PLC, as well as compile the resulting program. The main advantage of employing automation tools from UCPC include faster development, configuration and commissioning. In addition, using a common set of abstract process automation objects provides consistency of operation across applications.

There are two main motivating factors for developing a CI system for UNICOS-based control systems. Firstly, since the engineering of applications is performed using UNICOS objects, it is also natural that tests should be written at the level of these objects. Secondly, the UNICOS framework itself is written and maintained by CERN, so a CI pipeline which ensures that framework changes do not alter the behaviour of the objects or applications is crucial. Even though this work focuses on UCPC, the testing methodology based on OPC-UA is generic, and can easily be generalised to non-UNICOS PLC applications.

PLC TESTING

OPC-UA provides access to variables in the PLC memory, during program execution. Essentially, a client is shown a tree representation with the variables of a PLC program and their respective symbolic names. The structure of this tree varies slightly across different PLC types (e.g. Siemens S7-300/400/1500 or Schneider). Some of these changes are due to OPC-UA server to which they connect, others are due to implementation details of the UNICOS framework. For instance, inputs and outputs are split in an additional hierarchic level per each UNICOS object in S7-1500, comparing to S7-300/400. However, these differences can easily be accounted for, and we can thus envision a unique test for a given UNICOS application specification, independent of the underlying PLC type or low-level implementation. An additional advantage of using the OPC-UA interface is that it

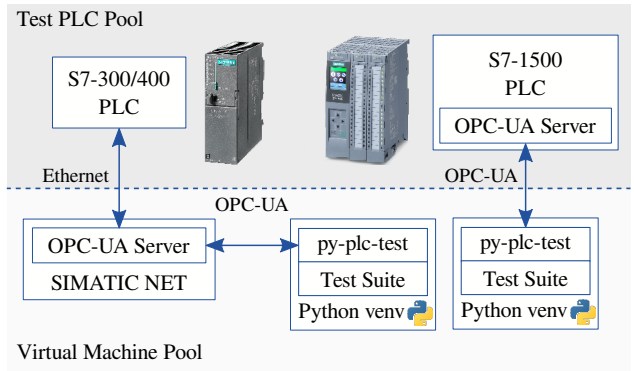

Figure 1: Testing architecture, with external (S7-300/400) and onboard (S7-1500) OPC-UA servers.

allows us to interact with PLC variables with minimal intrusion in the program (typically the only modification required is to disable the periphery addressing, to allow the input variables to be written by OPC-UA without being overwritten by the PLC scan).

In order to fulfil these requirements we created a Python package which we named `py-plc-test`. Python was chosen due to its readability and flexibility, which translates in ease of creation and improved maintainability of tests. This novel framework is intended to be used as a dependency and does not itself implement any test-related classes or business logic. It abstracts from the structural differences in the OPC-UA tree exposed by different servers, by having a predefined mapping for each of the supported server types.

We use the OPC-UA client API from open source LGPL Pure Python OPC-UA[1] package. Note that OPC-UA is asynchronous w.r.t the PLC cycle time, i.e no effort is made to synchronise reading from and writing to communication buffers at a particular part of the CPU cycle. However, the Python client ensures each operation has been communicated with the server. So far we have demonstrated our method using Siemens S7-1500 PLCs, which have an onboard OPC-UA server, and S7-300 which can connect to a SIMATIC NET OPC-UA server in a separate workstation, via Ethernet.

Figure 1 shows a schematic representation of the testing architecture. PLCs are physically located in a test laboratory, whereas the test runners and external OPC-UA servers are run in openstack[2] virtual machines.

UNICOS Abstraction Layer

By incorporating domain-specific features in our testing framework we can write simpler, concise and overall more readable and maintainable tests. Rather than accessing memory values by symbol names, we provide a higher layer of abstraction, at the UNICOS object level. Essentially, we allow users to refer to high-level objects, each type corresponding to its own Python class. Therefore, we can map a

[1] python-opcua GitHub repository as of October 9, 2019
[2] openstack webpage, as of October 9, 2019

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WESH4003

Table 1: High-level Command for UNICOS Field Object with Operational Modes

High-level	Internal implementation	Low-level value changes
valve.set_mode("manual")	valve.set_attribute("AuIhMMo", False)	VALVE.AuIhMMo = False
	valve.reset_register("ManReg01")	VALVE.ManReg01 = b0000 0000
	valve.set_attribute("ManReg01.MMoR")	VALVE.ManReg01 = b0000 0010

set of OPC-UA nodes to a Python class, and provide methods to perform more complex operations. Table 1 shows a simple example of a high-level command for a UNICOS field object (e.g. valve) with operational modes.

This UNICOS layer was built with the fail-fast principle in mind. As soon as an inconsistency is detected a specific error should be presented to the user so that the underlying problem can be quickly detected and resolved. For instance, it is trivial to detect a change that accidentally breaks a particular field in an object, by renaming it or changing its data type. We wrote unit tests for the framework itself, to ensure it is consistent with the latest specification of UNICOS.

CONTINUOUS INTEGRATION PIPELINE

We propose a new method for continuous integration of PLC projects, resorting to GitLab CI pipelines. Our pipelines consist of three stages, namely:

1. Logic and instance generation;
2. PLC project generation;
3. Deploy and test.

The stages are executed sequentially, but not necessarily performed by the same worker, as long as the required artefacts from previous stages are passed down the pipeline. Presently, we assume that each worker machine is connected to a single test PLC. This assumption allows us to leverage the default job scheduling of GitLab runners. Concretely, we wish to avoid situations where a test suite is running for a given PLC, and another runner is somehow able to download a new program onto it, thus breaking the tests.

Logic and Instance Generation

The project generation is handled by UNICOS Application Builder (UAB) [4]. This tool is responsible for creating the code for the PLC program, by using the information of the UNICOS objects and the process knowledge, stated in the project specification file. UAB can easily be configured to automatically perform build tasks through Maven[3]. These include the instance, logic and project generator plugins for each of the supported PLC types. The instance and logic generator plugins automatically produce a set of PLC source files (structured text), describing the UCPC object instances, the symbol table with respective address mapping and the application-specific control logic.

Whereas the creation of PLC source files can be performed solely by UAB, the compilation of these sources into the resulting PLC program must generally be done using the respective proprietary engineering tools.

We also use Maven to generate the files required for importing the objects' information to the SCADA project. Specifically, the SCADA system we employ is WinCC OA[4].

PLC Project Generation and Deployment

In our work, we focused on supporting Siemens S7-300/400 and S7-1500 series PLCs, as they are widely used in CERN's control systems, and both currently provide a reliable solution for OPC-UA communication. The engineering tools for S7-300/400 and S7-1500 PLCs are respectively SIMATIC STEP 7 and TIA (Totally Integrated Automation) Portal.

SIMATIC STEP 7 STEP 7 provides Visual Basic and C# APIs which can be used to create and edit projects. We build on previous efforts to automate STEP 7 project building [5] and add features that allow us to compile and download programs to their respective stations. Both the test PLC and the machine running the SIMATIC NET OPC-UA server it connects to are stations. This machine is not required to coincide with the one that handles the compilation steps.

TIA Portal The process of building a TIA Portal project through UAB uses mostly TIA Openness Scripter, as an alternative to the C# TIA Openness API. This choice was mostly propelled by the development cycle of TIA Portal, which resulted in frequent and numerous API changes. However, at the time of writing it was not possible to download a program to a PLC using the scripter. As a result, a small custom tool command-line tool was written which uses the C# API to enable downloading to a PLC.

Deploy and Test

Once the PLC project has been compiled it is then automatically deployed to an available testing target. We merge the deploy and test jobs in order to ensure the test suite is configured to target the correct test PLC. The network connections can be configured at the build stage and its details passed down the pipeline as artefacts. Alternatively, the configuration can be performed immediately prior to deploying to the test PLC, by resorting to the engineering tools' respective in-house command-line utilities.

Our current solution pre-configures the network interfaces in a base project, of which there is one per runner machine. The base project contains a PLC with a specific network

[3] Apache Maven, as of October 9, 2019

[4] WinCC OA webpage, as of October 9, 2019

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WESH4003

configuration to which we import the sources generated by UAB. In the case of STEP 7 projects, the SIMATIC NET OPC-UA server and its connection to the PLC is also configured in this base project.

We propose that tests are written in an imperative fashion, setting input values or trigger high-level commands, wait for the changes to propagate and then evaluating a set of outputs. Our framework is built to support this methodology. Even though OPC-UA supports asynchronous programming paradigms, we have yet to develop tests involving subscriptions to value changes.

Single Test Description We suggest a simple paradigm for developing test suites which can easily be extended to support different PLC types, without having to rewrite test business logic. Each set of tests has a corresponding test case class. Each concrete test case inherits from a generic test base class. We write the body of the test assuming a generic PLC object, and use it to obtain UNICOS objects by their name. One can then operate on these instances, and the changes are communicated to the real PLC via OPC-UA.

We create a class per each of the PLC types supported in our framework, which overload the default test case `setUp` and `tearDown` methods. By exploiting multiple inheritance in Python we can then associate a concrete test case with each of the supported PLC types, thus avoiding code duplication. These subclasses have access to the core logic, inherited from the concrete test, and the required methods to communicate with the desired PLC from the helper class.

Figure 2 shows the respective class diagram.

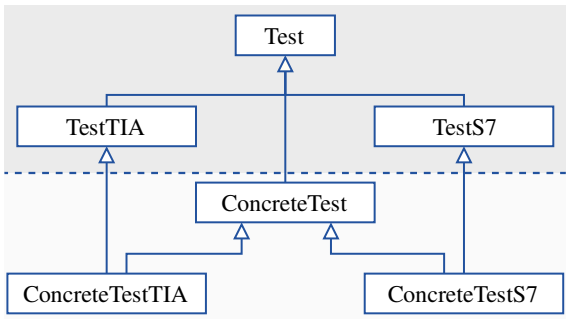

Figure 2: Class diagram for suggested test suite architecture.

UNICOS FIELD OBJECTS TEST

We applied our new method and software tools to a set of existing tests, meant to verify the correct behaviour of so-called UNICOS field objects. These include devices for representing process equipment driven by a digital and/or analogue signals such as pneumatic valves and control heaters.

The existing tests required multiple instances for each of the framework objects with different configurations [6] so as to mimic the behaviour of devices corresponding to real instruments, namely valves, motors and pumps. This limitation was due to the impossibility of changing an object's configuration at runtime through the SCADA. Additionally, the tests needed many additional devices to be created in order to excite the inputs and evaluate the outputs of the objects under test. Contrastingly, our approach allows us to test each object in isolation.

The tests were initially defined in a spreadsheet, which specified the sequence of steps (i.e. input changes) and the respective set of expected outputs per object. A code generator would extract the test description from the spreadsheet and automatically produce the test scripts, which operate on the SCADA stack.

Due to the auto-generated nature of the resulting test code, the output is compact but unintuitive. The original spreadsheet specification is not descriptive of each of the behaviours we intend to test, as it specifies long sequences of steps which are expected to produce different results depending on the possible object configurations. As a result, the purpose of the test, i.e. the expected behaviour of an object in a given situation is not evident and lost in the test description. This essentially makes the tests hard to understand, and even harder to update and maintain over time.

OPC-UA gives us direct access to every object input and output, as well as the configurable parameters, allowing us to reconfigure a single instance per object type according to each specific test. Because of this, our approach is able to greatly reduce the total number of required UNICOS object instances in the project.

We implemented these tests in Python, resorting to our novel framework. We follow the methodology provided in the previous section. It allows us to create a single test description and produce test classes that support both Step 7 and TIA Portal projects, requiring no modification to the concrete test logic. Perhaps the major improvement over the previous description is that it is self-contained, contrasting with the previous implementation; to fully understand a test one would have to first be familiar with the project specification file, as it describes the configuration of each test object instance, then the testing routines themselves and finally parse the matrix of expected values, for each step of the routines, specified in yet another spreadsheet. Listing 1 features part of a test for a specific UNICOS object type.

We then integrated these tests in an end-to-end pipeline, for both STEP 7 and TIA Portal, from UCPC project specification file to unit test results. Essentially, we automatically trigger the generation of the PLC project, deploy it to target S7-300 and S7-1500 PLCs and run the Python test suite.

Assumptions and Limitations

Presently, we assume that each runner machine has its dedicated PLC to prevent destructive operations. This allowed us to leverage the out-of-the-box scheduling features of GitLab CI instead of writing our own scheduler. However, this may pose an obstacle to scalability in the future.

Furthermore, we had to patch the logic generator template libraries of UAB due to implementation details. Essentially, the UNICOS objects are never supposed to be completely detached from I/O objects, and if no connection is defined in the specification file, the default behaviour is to hard code a

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WESH4003

```python
def test_orders_fs_off(self):
    """ORDERS - AnaDO with FailSafe Off"""
    anado = self.plc.get_anado(TARGET_ANADO)
    anado.configure(fs_pos_on=False, ...)

    self.set_mode_assert(anado, "manual")
    anado.set_status(False)
    anado.set_input_request(0.0)
    sleep(OUTPUT_DELAY)
    self.assertEqual(False,
        anado.get_attribute("OutOnOV"))
    self.assertEqual(0.0,
        anado.get_attribute("OutOV"))
    ...
```

Listing 1: Example test for AnaDO object from refactored UNICOS field objects tests.

constant value that is assigned at each cycle. This renders our attempts to change these variables via OPC-UA pointless, as the value is immediately overwritten by its default value. We were therefore required to work with our slightly modified version of UAB.

CONCLUSION

In this work, we present an alternative approach to PLC testing which relies on OPC-UA, thus removing the need for the SCADA stack. Concretely, we develop a novel framework which we use in a refactored version of existing tests for UNICOS objects. These tests were specified in a rigid fashion, which made them harder to modify and maintain. Moreover, they required adding many helper devices which greatly increased the complexity of the testing project. Our solution allows us to write a simple but powerful test description at a high-level abstraction layer, decoupling the implementation from the type of PLC under test.

Our contributions can be summarised as follows

1. A Python package which allows us to communicate with a real PLC via OPC-UA and benefit from a UNICOS abstraction layer, i.e. aggregate a set of scattered OPC-UA nodes and treat them as a high-level UNICOS object;

2. A unified description of tests for UNICOS field objects, which supports both STEP 7 and TIA Portal PLC projects;

3. GitLab CI pipelines to automatically build and test the PLC project for both STEP 7 and TIA Portal projects. These required creating utilities that use the proprietary tools' API, mostly for deploying the programs to the testing PLCs.

Future Work

We predict a vast number of possible use-cases for this testing methodology and recognise room for improvement in our implementation. Firstly, we should experiment with non-UNICOS projects, and improve our tool by including high-level features for generic PLC projects.

We have assumed our tests can be made of simple set, wait and assert routines. In the future we might benefit from implementing support for asynchronous communication, allowing for subscriptions to OPC-UA node value changes. This could eliminate the need for explicit waiting periods in our tests.

Regarding the pipeline, we may be able to perform the network configuration of a project in the last stage, right before deployment. This would increase the process' flexibility by reducing our dependency on manually-configured base projects. Additionally, we could optimise resource management, by writing our custom scheduler for coordinating the deployment and test stage. For instance, we could move the testing to a Linux machine with a Python installation, freeing a worker machine to build PLC projects.

Concerning CERN-specific use-cases, we highlight the possibility to test other components of UCPC by designing a set of test projects, such as the one implemented in this work. For instance, updates to the UCPC resource package could trigger a pipeline to run on each of the test project repositories, and be conditioned on whether they are successful or not. We have built a prototype for this use-case, by mimicking multi-project pipelines through the usage of GitLab remote pipeline triggers.

REFERENCES

[1] G. Sallai, D. Darvas, and E. Blanco, "Testing, simulation, and visualisation of PLC programs using x86 code generation," CERN, Tech. Rep. EDMS 1844850, 2017, http://edms.cern.ch/document/1844850.

[2] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC Unified Architecture*, 1st. Springer Publishing Company, Incorporated, 2009.

[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] B. F. Adiego, E. B. Vinuela, and I. P. Barreiro, "UNICOS CPC6: Automated code generation for process control applications," in *Proc. 13th Int. Conf. on Accelerator and Large Experimental Control Systems (ICALEPCS'11)*, Grenoble, France, Oct. 2011, pp. 871–874.

[5] C. Fluder, V. Lefebvre, M. Pezzetti, P. Plutecki, A. T. González, and T. Wolak, "Automation of the Software Production Process for Multiple Cryogenic Control Applications," in *Proc. 16th Int. Conf. on Accelerator and Large Experimental Control Systems (ICALEPCS'17)*, Barcelona, Spain, Oct. 2017, pp. 375–379.

[6] B. Fernández Adiego, E. Blanco Vinuela, and A. Merezhin, "Testing & verification of PLC code for process control," in *Proc. 14th Int. Conf. on Accelerator and Large Experimental Control Systems (ICALEPCS'13)*, San Francisco, CA, USA, Oct. 2013, pp. 1258–1261.