TESTING SOLUTIONS FOR SIEMENS PLC PROGRAMS BASED ON PLCSIM ADVANCED

Gy. Sallai*, D. Darvas, E. Blanco, CERN, Geneva, Switzerland

Abstract

Testing Programmable Logic Controllers (PLCs) is challenging, partially due to the lack of dedicated support for testing. Isolating a part of the PLC program, feeding it with test inputs and checking the test outputs often require manual work and physical hardware. The Siemens PLCSIM Advanced tool is a simulator solution for new generation Siemens PLCs and provides a rich application programming interface (API). This work presents a testing workflow for PLC programs built upon the capabilities of the PLCSIM Advanced API and the TIA Portal Openness API. Our tool takes a test case described in an intuitive but powerful tabular format, which is then executed on the full PLC program or a selected part of it. Outputs are captured automatically via the simulator API. Experience with this workflow shows that it offers an automated and scalable solution for PLC program testing and is applicable for multiple levels of testing without the need of using a physical hardware.

INTRODUCTION

Programmable Logic Controllers (PLC) are widely used devices for process control and automation. As their error-free operation is crucial, thorough verification and testing is required to gain confidence in their correct operation. While a plethora of tools are available for testing other languages and platforms, testing support for PLC programs is rather limited. As such, PLC program testing is often restricted to higher levels, with the need for a physical hardware and some tedious manual configuration steps.

This means that there is a need for improvement in PLC program testing. Testing should be easier (with easy configuration and less manual effort), more accessible (by reducing the need for dedicated testing hardware) and should not require the modification of existing source code.

Our solution, built upon Siemens PLCSIM Advanced, offers a scalable, accessible and user-friendly workflow for testing Siemens PLC programs. Using a simulator allows us to perform testing without having hardware in the loop, and by using the API offered by the simulator, the whole process can be automated. We describe a test table format, which is easy to understand and allows developers to conveniently define test cases. In addition, we introduce an automated process to isolate certain parts of the PLC program. This isolation allows us to perform lower-level (such as unit or integration) testing on PLC programs. As the whole process is automated, it may easily be used in conjunction with continuous integration solutions.

* Corresponding author. E-mail: gyula.sallai@alumni.cern

PLC PROGRAM TESTING

Testing is the process of verifying that a system is fit for its purpose and satisfies its specified requirements. For the remainder of the paper, we define three levels of program testing, relevant to our use cases.[1]

- *Unit testing* aims to test small, individual components of a system. For PLC programs, a practical unit-level component is a single function or function block. Unit testing a component requires the isolation of said component from its dependencies and dependant blocks.
- *Integration testing* targets the interaction of some units of the system, i.e. the interaction of different program blocks.
- *System testing* is performed on the whole application, checking whether the system satisfies its functional requirements.

In the domain of PLC programming, testing is usually done at the system testing level. The most commonly used testing procedures are *factory acceptance testing* (FAT) and *site acceptance testing* (SAT), as defined by the IEC 62381 standard [1]. FAT aims to verify system correctness by scrutinizing the production software on a (usually) dedicated testing hardware in laboratory conditions. SAT is done on the premises of the final installation, verifying that the software and the hardware equipment work together as intended.

Most testing workflows for PLC programs require physical hardware. However, testing with the involvement of physical hardware poses several challenges.

- It is hard to automate: hardware cannot be acquired, assigned, and configured on-demand.
- Feeding the inputs and capturing the outputs often requires manual effort, e.g. supplying inputs through the supervision system.
- Outputs cannot be extracted precisely – the latency induced by I/O communication makes it difficult to extract values at an arbitrary point of the program.
- For lower level (unit and integration) testing, precision and traceability often demands to check the behaviour of the program within really small time periods or at a given program location. However, there is no support in PLC hardware for such use cases.

[1] Unless indicated otherwise, this paper follows the terminology defined by the International Software Testing Qualifications Board (ISTQB). See `https://glossary.istqb.org` for further information.

Table 1: Feature Comparison of Various Simulation Solutions

	Translation to C	PLCSIM	PLCSIM Advanced
Access to I/Q/M	✓	✓	✓
Cycle-by-cycle execution	✓	✓	✓
Time synchronisation			✓
Automatable	✓		✓
Supported CPUs	All*	S7-300/400, S7-1200/1500	S7-1500 only

* Translation must be implemented for each input language. Semantic differences in execution may arise.

These challenges can be met by the exclusion of the physical hardware and using a simulator instead. A simulated PLC replicates most of the behaviour found in real hardware, offers more control and does away with some of the problems introduced by unpredictable communication. As software products, simulators can be spawned and used easily, with simpler and fewer configuration steps (e.g. no cabling).

SIMULATION AND TESTING BASED ON PLCSIM ADVANCED

Table 1 offers a feature overview of some simulator solutions available for testing Siemens PLC programs. We investigated simulation packages based on the following criteria.

1. Their ability to access memory, most notably the input, output, and global internal memory (I/Q/M) areas.
2. Support for cycle-by-cycle execution, i.e. the ability to step between PLC program cycles.
3. Time synchronisation capabilities, precise and controlled execution for a given (possibly short) timespan.
4. Automation capabilities, support for programmable simulation.
5. Supported target PLCs.

Our previous attempts on unit testing involved the translation of PLC code to C or Java [2]. This approach gave us the power of defining and executing tests efficiently in a well-known environment, with a lot of support from unit test libraries. However, due to the semantic differences between a general purpose programming language and one designed for PLCs, the translation lacked several features, most notably proper time synchronisation.

Another possibility is PLCSIM, a simulation solution integrated into the Siemens development environments.[2] PLCSIM offers semantically correct simulation, with capabilities for forcing particular values from the outside, but it lacks proper time synchronisation. Furthermore, it is a tool mostly designed for manual use, thus there is no easily programmable API to automate its execution.

[2] Not to be confused with PLCSIM Advanced, the target of this article. PLCSIM is the built-in simulation package found in either STEP7 v5.x or TIA Portal. PLCSIM Advanced is an optional software, largely independent of the "regular" PLCSIM.

PLCSIM Advanced is an optional simulator package for Siemens S7-1500 PLC series, which addresses the shortcomings discussed previously. It allows writing to and reading from arbitrary global memory locations (such as the I/Q/M areas of the PLC and contents of data blocks). It provides continuous, cycle-by-cycle and time-synchronised execution modes and comes with a C♯ API which can be used to automate the simulation and execution process.

Test Definition

We use a form of generalised test tables described by Weigl et al. [3], slightly adapted to our needs. In this test specification format, each column represents a *variable*, which might be a PLC symbol or memory location. Rows represent different *test steps*. Each test step has a defined *duration*, which might be a given number of PLC cycles or a timespan. An individual cell can either be an input value for input variables or an expectation for output variables.

Input values must either be fully defined or denoted as "don't care" with a dash (`-`) symbol. If an input value is set to don't care, the simulator will force no value for that particular input in the corresponding test step. An *output expectation* could be a concrete value, a complex constraint or a don't care. If an output cell contains a fully defined value (e.g. `0`), the captured output value of the corresponding variable must be equal to this particular value. In addition, we allow more abstract constraints in the cells. Output cells can contain expressions such as `>0`, which are constraints to the actual value of the corresponding variable (i.e. the captured value for a given PLC variable in the given test step must greater than zero). We can also specify *range constraints* such as `1..5` where the user can define a range of acceptable values. If the output constraint is a don't care, the actual output is ignored and the output expectation is always satisfied.

Example. Table 2 shows an example test table. The first row lists the different PLC variable names, while the second row declares the type of each column. Assume that the variable `B` is an *inout* parameter. Thus, it is present in the table twice: once as an input and once as an output. During execution, the test runner first forces the values present the input column for `B`, then after the specified test step duration, it asserts against the values in the output column.

While test tables offer a powerful, precise, easy-to-understand format, they may prove to be too low-level for larger applications. Due to the lack of any abstraction, cer-

Table 2: An Example Test Table

A	B	X	Y	B	
input	input	output	output	output	duration
True	1	-	>X	5	80ms
False	4	-	5	2	10
False	-	1..5	>(X + Z)	-	200ms

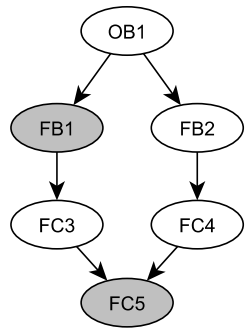
tain high-level commands which set multiple values (or bits in different registers) may be inconvenient to represent. However, it is possible to build upon our format and extend with a component, which is able to read a high-level test case files and generate the low-level test tables required by our tool.

Test Project Generation

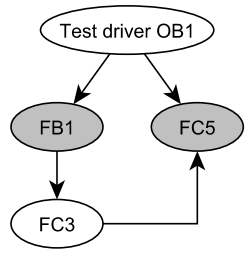
Using our solution, system testing requires no modification in the source code of the user program. For unit and integration testing, however, the unit under test (UUT) needs isolation from its callers and dependencies. In addition, a test driver is needed to invoke the test process for each UUT. While general purpose programming languages offer quite a lot of support for unit isolation and test drivers (separate binaries, test runners, mocking frameworks, etc.), there is no dedicated support for such features in the PLC domain. As the simulator can only execute programs starting from a single entry point (the entry OB), adding a test driver would imply unacceptably intrusive and tedious changes on the user program. We overcome this issue by generating a new PLC project for unit testing, based on the original program, using TIA Portal Openness.

Consider a PLC program with the call graph (meaning that `OB1` calls `FB1` and `FB2`, `FB1` calls `FC3`, and so forth) shown in Figure 1a. The test project generator creates the unit test project for a given set of *criteria blocks*, i.e. the blocks we wish to isolate. In this example the criteria blocks are `FB1` and `FC5`, shown with a filled background. The test project generator first strips away all blocks which are unneeded to execute the criteria blocks (in this case `FB2` and `FB4`). In the second step, it generates special data blocks, which will be used by the test executor to interact with the data of the UUTs. Finally, a new test driver OB is generated, which will serve as the entry point of the program, calling all program blocks marked for isolation. Figure 1b shows the call graph obtained after this procedure. This test driver uses a separate identifier (a hash value) to distinguish between program blocks. Clients may write this variable (`"TEST_DRIVER".TestSelector`) to decide which program block to test in a particular test case. Figure 1c shows this generated test driver code for our current example.

Note that this process only isolates blocks from its callers by calling them explicitly from the generated test driver. Isolating blocks from their dependencies requires more sophisticated methods, such as mocking. We are currently investigating the problem of providing meaningful mocks for PLC programs in a non-intrusive way.



(a) Call graph of the source PLC project.



(b) Call graph of the project generated from Figure 1a.

```
CASE "TEST_DRIVER".TestSelector OF
    16#49ca240b: "DB_FB1"();
    16#497d39ca: FC5(i := "DB_FC5".i, q => "DB_FC5".q);
END_CASE;
```

(c) The generated test driver code.

Figure 1: Test project generation for a simple PLC program.

Test Execution

PLCSIM Advanced offers two execution modes relevant to our case. First, a time-synchronised mode, in which execution lasts for *at least* a given timespan. Second, a cycle-by-cycle execution mode, which lasts for one PLC cycle and may be used to step between cycles. In both cases, the simulator only allows execution to freeze on so-called *synchronisation points*. Such synchronisation points are hit before the execution of each PLC scan cycle. This ensures that we always read consistent values from the virtual PLC.

During execution, test steps are processed in the order the test case table declared them. At the beginning of each test step, we write the input values to the virtual PLC and start execution according to the value of the duration column. If the duration is specified to be a timespan, the simulator will run in time-synchronised mode for the requested time plus the time required to reach the next synchronisation point. If a number of cycles is specified as the duration, the simulator will run precisely for the given number of cycles. At the end of the test step, we extract the actual test outputs from the memory of the virtual PLC using the simulator API. Finally, the captured outputs are compared to the output constraints defined in the test case table, yielding a test verdict.

Test Report

Our tool is able to represent a test report in various formats, such as plaintext, HTML, timing or waveform diagrams. To obtain as much information as possible, the behaviour of the system is recorded in every cycle. This offers opportunities to display detailed reports to the user.

Figure 2 shows such a report for a hysteresis module with a configurable delay in the timing diagram format. Notice how the elapsed time signal is ramping up cycle-by-cycle

and how the output signal raises when the elapsed time hits the configured delay.

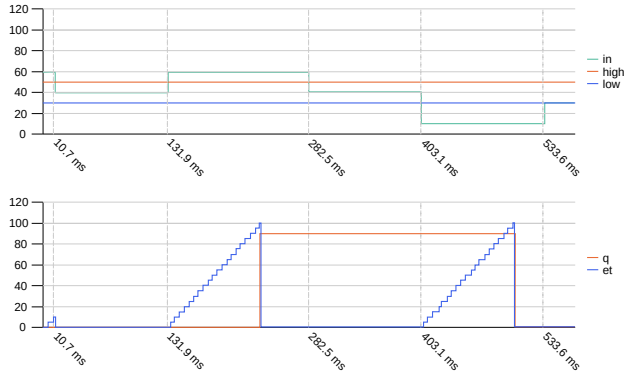


Figure 2: Timing diagram report for a hysteresis module with a configurable delay. The chart on the top shows input signals, the bottom chart shows the outputs.

Test Automation

In order to support full test automation, our tool is able to automatically compile and download the user program to a freshly spawned virtual PLC via TIA Portal Openness. Through our tool's command line interface, the whole process can be executed in a continuous integration (CI) pipeline, such as Jenkins or GitLab CI. This allows users to have automatically executed unit tests on each commit. As the tool is able to generate test reports in JUnit's XML format, the (possible failing) test results can be shown natively in GitLab's interface on each merge request. Furthermore, automated execution offers a convenient way to execute tests on a remote machine, thus reducing the need to install the simulator on every developer's workstation.

USE CASES

The unit and integration testing capabilities of our tool have already been demonstrated in two real-world instances. We are currently evaluating our testing workflow on the system testing of a full-scale application.

Unit Testing UNICOS Baseline Objects

The UNICOS-CPC [4] framework ships with a handful of baseline objects, the building blocks of a UNICOS application. As such, proper unit testing for these objects is an important issue. We have built a prototype testing workflow for UNICOS baseline objects, partly translating our previous test cases into test tables. This workflow uses our test project generation feature, along with automated execution through GitLab CI.

Reproducing Counterexamples from Formal Verification

PLCverif [5] is a formal verification tool for PLC programs, developed at CERN. It supports several underlying model checking engines, which systematically explore the state space of a program to provide a proof or counterexample of a given requirement property. If formal verification reports a property violation, the resulting counterexample is usually time- or cycle-sensitive, meaning that the violating execution is hard to reproduce using traditional testing methods. However, PLCverif counterexamples can be represented in our test table format, providing a suitable input for our tool, which can execute it in a time- or cycle-synchronised manner. This allows developers to easily turn formal verification results into reproducible test cases. We are currently using this approach for the verification of the CERN SPS accelerator Personnel Protection System.

CONCLUSION

In this paper we presented our PLC program testing solution based on the PLCSIM Advanced simulator, its C♯ API and TIA Portal Openness. We described an intuitive tabular test definition format, which can be executed using a simulated PLC. During execution, our tool feeds inputs, captures outputs, and provides a high-resolution recording for test reports through the simulator. This allows automated testing, with great precision, but without the need for human intervention or physical hardware. In order to facilitate unit testing with our testing framework, we proposed a solution to (partially) isolate program blocks for unit testing by generating a new unit test project from a PLC program. The tool was built to be high-level and easily extensible, thus it is possible to adopt it to other publicly available and suitable simulators for other vendors and PLC series. Our workflow has demonstrated its usability in multiple testing scenarios, showing that it provides an easy-to-use, automated, and accessible continuous testing solution for PLC programs to reveal errors as early as possible.

REFERENCES

[1] *IEC 62381:2012 Automation systems in the process industry – Factory acceptance test (FAT), site acceptance test (SAT), and site integration test (SIT)*, IEC Standard IEC-62 381, 2012.

[2] G. Sallai, D. Darvas, and E. Blanco, "Testing, simulation, and visualisation of PLC programs using x86 code generation," CERN, Tech. Rep. EDMS 1844850, 2017.

[3] A. Weigl, F. Wiebe, M. Ulbrich, S. Ulewicz, S. Cha, M. Kirsten, B. Beckert, and B. Vogel-Heuser, "Generalized test tables: A powerful and intuitive specification language for reactive systems," 2017, pp. 875–882.

[4] unicos-cpc, `http://unicos.web.cern.ch/unicos-cpc`

[5] E. Blanco Viñuela, D. Darvas, "PLCverif Re-engineered: An Open Platform for the Formal Analysis of PLC Programs", presented at the *17th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'19)*, New York, USA, October 2019, paper MOBPP01, this conference.