Prototype of Automated PLC Model Checking
Using Continuous Integration Tools
CERN Summer Student Report

Michael Lettrich\textsuperscript{1}, Daniel Darvas\textsuperscript{2}, and
Borja Fernandez Adiego\textsuperscript{3}

\textsuperscript{1}CERN, m.lettrich@gmail.com
\textsuperscript{2}Supervisor, CERN, daniel.darvas@cern.ch
\textsuperscript{3}Supervisor, CERN, borja.fernandez.adiego@cern.ch

September 25, 2015

Abstract

To deal with the complexity of operating and supervising large scale industrial installations at CERN, often Programmable Logic Controllers (PLCs) are used. A failure in these control systems can cause a disaster in terms of economic loses, environmental damages or human losses. Therefore the requirements to software quality are very high. To provide PLC developers with a way to verify proper functionality against requirements, a Java tool named PLCverif has been developed which encapsulates and thus simplifies the use of third party model checkers. One of our goals in this project is to integrate PLCverif in development process of PLC programs. When the developer changes the program, all the requirements should be verified again, as a change on the code can produce collateral effects and violate one or more requirements. For that reason, PLCverif has been extended to work with Jenkins CI in order to trigger automatically the verification cases when the developer changes the PLC program. This prototype has been used in real-life PLC programs at CERN and its potential have been recognized.
1 Introduction

With its accelerators and complex experimental facilities, CERN, the European Organization for Nuclear Research, has become a driving force in the development of new technical solutions in the past decades. The challenges imposed by the complexity, scale and required precision of new technical solutions are increasing with each new project. For many experiments and central systems (e.g. cooling and ventilation or cryogenics) it thus becomes necessary to introduce industrial automation technologies on a large scale to control and monitor such systems. These control and automation tasks are often tackled using so-called Programmable Logic Controllers (PLCs), dedicated, hard real-time systems, which are programmed by domain-specific programming languages as specified by the IEC 61131 standard [1]. This hardware is also used in critical settings, where malfunctions can endanger people or equipment. Currently in industry, it does not exist a solution or commercial tool based in formal methods to guarantee that PLC programs meet the specified requirements. Good specification and testing strategies exist to minimize the amount of software bugs, however CERN also evaluates how to incorporate formal verification and apply it to PLC code to improve software quality.

2 On Formal Verification of PLC Code

*Formal verification* mathematically proves or disproves if an algorithm is correct or incorrect with respect to a certain requirement. In industrial control system design, these requirements can be extracted from the specification of the system. As formal verification by hand is for most algorithms a neigh infeasible task, it has always been desirable to have formal verification performed by computers, which is still an ongoing task. As long as the problem can be expressed as finite state automata, *model checking*, an approach to formal verification can deliver excellent results as it provides exhaustive checking of possible input combinations and in case of a requirement violation is even able to give a concrete counter example with the assigned variables [2]. In the past years powerful model checking software has been developed and is slowly finding its way into industry. The above mentioned benefits make it a good complement for testing to discover non obvious bugs.
3 PLCverif for Formal Verification of PLC Code

While there are numerous model checkers, there is no interface that offers control engineers a way of directly verifying PLC code without a steep learning curve. Therefore PLCverif\(^1\), a Java tool based on Eclipse\(^2\) technology that encapsulates third party model checkers (currently NuSMV\(^3\) and nuXmv\(^4\)) was started. It currently is a prototype showing promising results.

PLCverif requires the PLC program and requirements to be checked as inputs. Requirements are formulated using patterns specified in natural language as a simplification of temporal logic. After parsing the PLC code and the requirements, an automaton intermediate model (IM) is built and different kind of reduction algorithms are applied iteratively to eliminate states and thus shrink the model size. This IM is then transformed into a specific model that is input model for the model checker. Finally the output of the model checker is parsed by PLCverif again to generate reports that are meaningful to the user (see Figure 1). For a detailed explanation see [3].

4 Continuous Integration Tools for Automated Verification

PLCverif demonstrates that parsing, reduction and model generation from PLC code can be automated. However to have a positive effect on code quality and development time, it is also necessary to integrate execution of

---

\(^1\)http://cern.ch/plcverif
\(^2\)http://www.eclipse.org
\(^3\)http://nusmv.fbk.eu/
\(^4\)https://nuxmv.fbk.eu/
the verification cases into the developers’ workflow.

As even minor changes in code might invalidate the results of all verification cases, it is necessary to execute all verification cases again after each change in the code. This way developers can make sure that software still works according to specifications or if not, are able to identify and fix bugs faster. However with the current tools, verification cases have to be executed one by one under human supervision. As verification is computationally and memory intensive, both human and computational resources are blocked for a non-neglectable time (the verification of the safety logic in CERN’s SM18 magnet testing facility e.g. consists of over 20 verification cases taking over an hour in total). As a result, the available verification cases are executed less often which decreases the developers’ benefits. To address this shortcoming, we have taken tools and workflows from continuous integration (CI), a technique frequently used in off-the-shelf software development – but not yet in PLC software development – and provided a prototype adapted to our needs.

4.1 Jenkins CI for Formal Verification

A CI system performs multiple building, testing and release actions in an automated fashion to improve software developers’ workflows. At each commit to a versioning system (SCM), building software, unit-testing of the build as well as calculation of code metrics are automatically applied and occurred
issues are reported to developers. For our prototype we rely on Jenkins CI\textsuperscript{5}, which is one of the most popular CI solutions available. It is a highly flexible solution with a large user base, under a open source license and is internally supported by CERN IT department.

To suit our needs we had to diverge from the original concepts of CI and replace building and unit-tests by verification. Users commit both code and verification cases to SCM. This SCM is then scanned for changes in fixed time intervals. If a change is detected, source code is verified against the given verification cases and status is then reported to the users (see also Figure 2). The aggregation of all commits in a certain time frame is necessary to avoid the creation long backlogs, caused by the computational complexity of verification.

### 4.2 Challenges Imposed by System Architecture

Jenkins CI is based on a master/slave architecture. Jobs are defined on the master node using a web interface where we define which SCM repository to scan, when to execute the job, which commands to execute and how to represent the results. Attached to the master node are slave nodes. These slaves perform the actual tasks specified in the job, once being triggered by the master and report their results back. As of technical restrictions at CERN, we were only able to bind Linux clients via SSH to the Jenkins master. An overview of verification runs are stored in the form of adapted JUnit XML. On the master node, the XML-encoded results are then processed into human-readable statistics and an email to the developers about passed and failed verifications is sent. Detailed information is also generated and presented to the user as an HTML report.

All verification jobs have a very similar structure. Thus we decided to create a generic job template from which then new, specific jobs can be created to further facilitate administration effort. However, the general system architecture and the restrictions described above raise the need for new software and new processes. A port of PLCverif to Linux is required as well as a command line interface, that can be used by scripts that are run by Jenkins.

### 4.3 Implementation of a Command Line Interface for PLCverif

PLCverif is based on Eclipse, a graphical integrated development environment (IDE) and therefore is mostly operated in graphical mode. But there

\textsuperscript{5}https://jenkins-ci.org
are no restrictions to provide a command line interface and running Eclipse in headless mode. To embed it into the current project structure a new plugin project containing an Eclipse product have been created. This allows to export a dedicated command line version of PLCverif that is built independently of the graphical version.

While internal core functionality of PLCverif could be accessed by a previously created API, the input and output interface of the headless version had yet to be created. As PLCverif as well as the CI integration are prototypes, changes in functionality are expected over time. Therefore both input and output interface had to be implemented in a modular fashion and with extensibility in mind. This was achieved by the use of design patterns and a high level of abstraction.

Invocation of the headless version should be flexible enough to be useful for both humans and scripts. We chose a simple yet powerful syntax:

```
PLCverif --in <verifCase1> ... --out <outputDir>
PLCverif -dir --in <inputDir> --out <outputDir>
```

An especially important requirement is to provide output based on the use case. While humans expect status reports and progress indication on the command line as well as simple, human readable output, verbose execution is often not desired in automated execution. Also results usually have to be represented such that they can be processed by other software. We thus implemented a modular output structure. Modules are enabled and disabled in a configuration file and are initialized upon project start.

### 4.4 Porting PLCverif onto Multiple Platforms

As mentioned in sub-section 4.2, due to technical restrictions we could only use Linux slaves. As a result, we were required to port PLCverif to Linux. As PLCverif is entirely written in Java and is based on Eclipse, we assumed that porting to multiple platforms with reduced effort is possible. Careful examination of the source code and Eclipse documentation\(^6\) confirmed the assumptions. As the used third party products (see section 3), are also available for the major platforms, we decided to port to PLCverif to all major platforms: Windows, Linux and Mac OS X for both the UI and headless version.

\(^6\)https://wiki.eclipse.org/Building
4.5 Semi-Automated Deployment of PLCverif

Eclipse provides tools and packages to export builds for multiple platforms. However, third party tools and configuration files still have to be included before being released. Also due to heterogeneous file systems in CERN infrastructure, executable rights for releases on UNIX-like systems may get lost. To tackle such problems, it is often advised to use build management tools like Apache Maven. At this early project stage however, we decided against build management for simplicity. Instead, it was decided to make releases internally available on CERN DFS, which can be accessed by CERN users via SMB or WebDAV protocol. Builds that are exported by developers of PLCverif from Eclipse are then processed by a shell script run from a Jenkins Linux slave and transformed and packaged to a final release.

See Figure 3 for further details. This way we can facilitate the set up for each platform and relieve the developers from the error-prone task of manual assembly of releases.

As the individual Linux slaves that run verification tasks when dispatched by Jenkins CI should always use the latest release available, each of them mounts CERN DFS via SMB and obtains the latest release prior to performing the verification tasks.
5 Outlook and Conclusion

With PLCverif, CERN has developed a tool showcasing that formal verification of PLC programs is possible and the complexity of using formal methods can be completely hidden from the developers.

In this project, the ideas of continuous integration from off-the-shelf software engineering and the ideas of formal verification have been combined and PLCverif was integrated in workflow of PLC programs development. This improves significantly the PLC code quality without additional burdens for developers. Currently PLCverif and the CI capabilities implemented in this project are prototypes, but the promising experimental results on real-life PLC programs give a positive feedback to the project in order to provide a production ready version of the tool.

References


Acknowledgement

I would like to thank my supervisors Daniel Darvas and Borja Fernandez Adiego for their enormous effort, support and time. I have learned many valuable lessons on software engineering, prototyping and time management in research oriented settings during the internship.