PLCverif: A TOOL TO VERIFY PLC PROGRAMS BASED ON MODEL CHECKING TECHNIQUES

D. Darvas*, B. Fernández Adiego, E. Blanco Viñuela, CERN, Geneva, Switzerland

Abstract

Model checking is a promising formal verification method to complement testing in order to improve the quality of PLC programs. However, its application typically needs deep expertise in formal methods. To overcome this problem, we introduce PLCverif, a tool that builds on our verification methodology and hides all the formal verification-related difficulties from the user, including model construction, model reduction and requirement formalisation. The goal of this tool is to make model checking accessible to the developers of the PLC programs. Currently, PLCverif supports the verification of PLC code written in ST (Structured Text), but it is open to other languages defined in IEC 61131-3. The tool can be easily extended by adding new model checkers.

INTRODUCTION AND MOTIVATION

Operating an accelerator complex to provide facilities for particle physics research involves numerous process control tasks. Many of them (such as cooling and ventilation, cryogenics, gas systems) are controlled by Programmable Logic Controllers (PLCs) at CERN, the European Nuclear Research Organization. As these systems are critical for the operation of CERN, the correctness of the executed PLC applications is a high priority.

Testing is a widely used solution to find potential failures in software. However, testing is not an universal solution for the verification of programs, for the following main reasons:

- Testing cannot show the absence of bugs, it can only show their presence.
- Testing can only check the outputs given by the software under test for some *selected input sequences* (test inputs). It cannot check efficiently general statements (e.g. "If output `FireAlarm` is true, then the output `NoAlarmPresent` should always be false.") or liveness properties (e.g. "If a request is received, a response will be sent *eventually*.").

Model checking is a good candidate to *complement* testing in order to reduce these weaknesses [1]. This paper introduces the high-level concepts of model checking and our proposed solution to incorporate model checking in the PLC software development process.

CHALLENGES OF MODEL CHECKING

Model checking is a formal verification technique that takes (1) a mathematical *model* of the system to be checked and (2) a formalized *requirement*. The model checker algorithms can decide if the given requirement is satisfied for the given model or not. Contrarily to testing, model checking checks *all possible* executions of the program and reports if any of them violates the requirement. Model checking is also able to generate *counterexamples*, i.e. input sequences demonstrating the violation of the given requirements.

However, model checking is not a silver bullet. There are three main obstacles of using this technique:

1. The model checker tools need a mathematical representation of the program. Constructing them needs lots of effort and experience in the formal methods domain.
2. The requirements should also be formalized for model checking. This is a similarly challenging task.
3. Model checking is computation- and memory-intensive. The generated models of the programs are often too large or too complex to be analysed with the available computation capacity.

These obstacles are difficult to overcome. The available tools require deep expertise in the formal verification domain. This could be the main reason why model checking is not widely used in industry yet, apart from some highly safety-critical applications in avionics, railway industry, etc.

Our goal is to provide a model checking solution for the PLC domain by overcoming the mentioned obstacles. All of them contain both theoretical and technical challenges. In earlier work [2, 3] we have provided solutions for the theoretical obstacles. All these solutions have been incorporated in a tool called PLCverif[1] that makes model checking accessible for the developers in the PLC domain. This paper focuses on this tool and on the bridge between the formal methods and PLC domains.

PLCverif: A BRIDGE BETWEEN FORMAL VERIFICATION AND PLC DOMAINS

In this section we overview the main features of the PLCverif tool. We focus on the user's point of view, therefore the structure of this section follows the normal workflow of a user.

Typical Workflow

The typical user workflow of the PLCverif tool consists of four steps:

1. Defining (importing or writing) the PLC code to be checked,
2. Defining the requirement to be verified,
3. Executing the verification,
4. Evaluating the results of the verification.

In the following sections, each step is described in detail.

* E-mail of the corresponding author: daniel.darvas@cern.ch

[1] `http://cern.ch/plcverif/`

Defining the PLC Code. The PLCverif tool provides an editor for PLC programs. The aim is to support the languages defined in the standard IEC 61131-3 [4]. Currently the tool supports SCL (Structured Control Language), which is the Siemens' implementation of the standard, high-level language ST (Structured Text). Additionally, SFC (Sequential Function Chart) and STL (Statement List) are partially supported. The code editor of the PLCverif (Figure 1) provides the main features required nowadays in modern development tools, e.g. syntax highlighting, content assist, support for refactoring. The PLC code to be verified can either be written in this PLC code editor, or imported if the program is already existing.

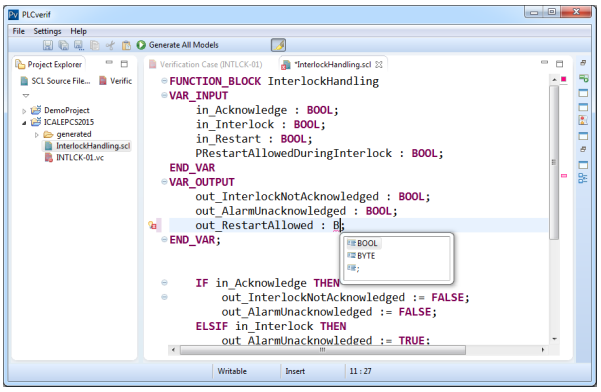


Figure 1: PLC program editor.

Defining the Requirement. Similarly to the test cases in testing, a *verification case* should be defined by the user. A verification case contains all necessary information for the verification. The user has to define:

- The general information of the verification case (ID, name, description),
- The source code to be checked,
- The requirement to be checked,
- The model checker tool to be used.

In addition, the user has the possibility to fine-tune the verification method by setting some parameters (e.g. the way the model is simplified by reductions). These parameters are set automatically using various heuristics. For example, the user can provide assumptions about the input variables (e.g. the value of an input variable is always false) to facilitate the task of the model checker, however this is optional.

The verification case can be edited on a form that can be seen in Figure 2.

Requirement patterns As previously mentioned, the requirement has to be formalised for the model checker. Typically, requirements are formalised using CTL (Computational Tree Logic) or LTL (Linear Temporal Logic). Both formalisms are difficult to be used for non-experts. Furthermore, even with experience it is easy to make semantic mistakes in the formalisation of requirements.

To avoid these issues, we have introduced a predefined set of *requirement patterns*, similarly to [5, 6], focused on the PLC domain. In our method, a requirement pattern is:

- A precise English sentence with gaps for logic expressions (e.g. "*If* $\ldots^1$ *(at the end of the PLC cycle), then* $\ldots^2$ *is always true (at the end of the same cycle).*"); and
- A formalization of the textual representation in LTL or CTL using the same gaps. (e.g. "AG $\left((EoC \wedge \ldots^1) \rightarrow \ldots^2\right)$", where *EoC* stands for end of cycle).

The task of the user is to choose a requirement pattern that corresponds to the requirement to be checked, and to fill the gaps of the pattern. Each gap has to be filled by an expression containing constants, variables, and logic operators (e.g. AND, OR).

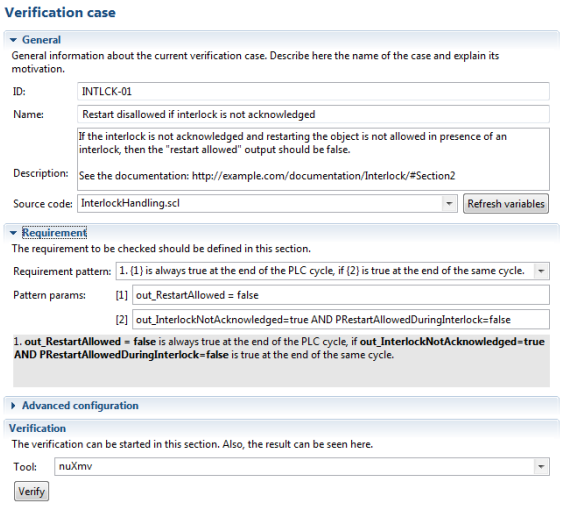


Figure 2: Verification case definition form.

Verification Process. After loading or writing the PLC code and providing the verification case, the verification procedure is fully automated.

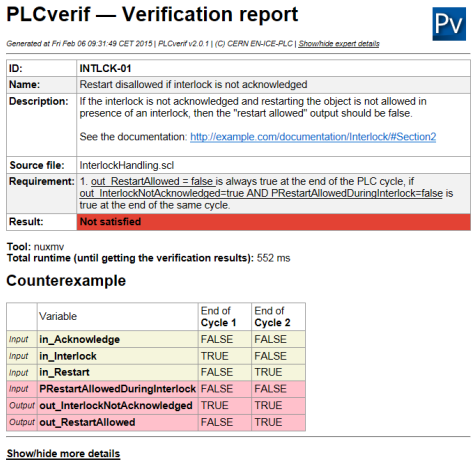
In the background PLCverif performs the following steps, completely hidden from the user:

1. The formal, mathematical requirement (in LTL or CTL) is produced based on the given information.
2. The PLC code is parsed and translated into an intermediate model.
3. The intermediate model is reduced using various techniques [3]. The goal of them is to simplify the model without modifying its meaning, and to remove any part of the model that is not necessary for the evaluation of the current requirement. Consequently, for each requirement a unique verification model is produced.
4. The reduced intermediate model and the given requirement is then converted to the input syntax of the selected model checker tool.
5. When the model and the requirement is produced, the model checker tool is invoked. All its outputs, including

the error messages are stored in the PLCverif tool to provide feedback to the user.

Evaluating the Results. The output of the model checker tools (the counterexamples) are typically difficult to understand. For example, the counterexamples can be huge, they have to be reduced before manual analysis. Also, the referred variable names can be different from the ones defined in the PLC code due to the automated generation process and the various restrictions of the formalisms. These have to be replaced by the names meaningful for the user.

The result of this phase is the verification report (see Figure 3 for example) — a self-contained document produced automatically that includes the details of the verification case, the result of the verification and the reduced counterexample, if applicable.



PLCverif — Verification report

Generated at Fri Feb 06 09:31:49 CET 2015 | PLCverif v2.0.1 | (C) CERN EN-ICE-PLC | Show/hide expert details

ID:	**INTLCK-01**
Name:	Restart disallowed if interlock is not acknowledged
Description:	If the interlock is not acknowledged and restarting the object is not allowed in presence of an interlock, then the "restart allowed" output should be false. See the documentation: http://example.com/documentation/Interlock/#Section2
Source file:	InterlockHandling.scl
Requirement:	1. out_RestartAllowed = false is always true at the end of the PLC cycle, if out_InterlockNotAcknowledged=true AND PRestartAllowedDuringInterlock=false is true at the end of the same cycle.
Result:	**Not satisfied**

Tool: nuxmv
Total runtime (until getting the verification results): 552 ms

Counterexample

	Variable	End of **Cycle 1**	End of **Cycle 2**
Input	**in_Acknowledge**	FALSE	FALSE
Input	**in_Interlock**	TRUE	FALSE
Input	**in_Restart**	FALSE	TRUE
Input	**PRestartAllowedDuringInterlock**	FALSE	FALSE
Output	**out_InterlockNotAcknowledged**	TRUE	TRUE
Output	**out_RestartAllowed**	FALSE	TRUE

Show/hide more details

Figure 3: Verification report.

Applied Technologies

The PLCverif tool is implemented as an *Eclipse RCP* application. This provides a standalone, multi-platform tool with an environment (look and feel) familiar to the users with moderate development effort. The PLC codes are parsed using *Xtext*. This framework provides editor, parser and an object model based on the defined grammar of the input language, that is SCL in our case.

Parsed programs are then translated to an intermediate model representation. This is a formalism conceptually close to the input of many model checking tools. This model is implemented using *Eclipse Modeling Framework*. The intermediate models are then translated into the concrete input syntax the model checker tools. The translation is developed using *Xtend*.

Currently the format of the NuSMV/nuXmv, UPPAAL and BIP tools are supported, but other, similar model checker tools can be easily integrated into PLCverif.

PLCverif has a command line interface too. It allowed us to set up a "continuous verification" workflow, where a *Jenkins* job automatically (re-)checks all verification cases on each SVN commit, then the results are sent in e-mail.

CASE STUDY

To illustrate our method, a simple example for the previously presented workflow is shown. Listing 1 shows a code excerpt, extracted from a base object of the UNICOS framework [7]. The original code is much larger, containing around 200 variables and 600 lines of code. The following steps correspond to the steps of the workflow presented before.

1. As a first step, the user writes or imports the example PLC code in Listing 1 to the tool.
2. The next step is the requirement specification. The example informal requirement is the following: *if the interlock is not acknowledged and restarting the object is not allowed in presence of an interlock, then the "restart allowed" output should not be true*.
   For this informal requirement a pattern should be chosen. The requirement pattern presented before is convenient to describe this requirement.
   Filling that pattern results in the following:
   > "If `out_InterlockNotAcknowledged`=true and `PRestartAllowedDuringInterlock`=false (at the end of the PLC cycle), then `out_RestartAllowed`=false is always true (at the end of the same cycle)."
3. Once the PLC code and the requirement are defined, the user should press the "Verify" button, every step is then automated. The model generation, the model reductions, invocation of the model checker and generation of the verification report takes less than a second in this case[2].
4. The verification report (Figure 3) shows that the requirement is not satisfied. A counterexample is also provided for the user. The counterexample can be seen in Table 1. Executing the code with the inputs shown in the table results that at the end of the second PLC cycle `out_InterlockNotAcknowledged`=true and `out_RestartAllowed`=true is possible, violating the requirement defined above.

After knowing that the requirement is not satisfied, using the counterexample, the violation can be reproduced by the developer. Investigation of the code can show the source of this problem: a a pair of parentheses is missing from the condition in lines 19–20 of Listing 1 (see the yellow marks). After fixing this issue the requirement will be satisfied.

This example highlights the main differences between testing and model checking:

- It is enough to define the requirement. Using testing, careful test planning and numerous test cases would have been necessary to check this requirement.
- After fixing the problem, model checking can even prove that the given requirement is satisfied. Contrarily, testing can only show the presence of bugs.

[2] The measurement was executed on a PC with Intel Core i7-3770 CPU, 8 GB RAM on Windows 7 x64. The selected model checker tool was nuXmv 1.0.1.

```
FUNCTION_BLOCK InterlockHandling
VAR_INPUT
       in_Acknowledge : BOOL;
       in_Interlock : BOOL;
       in_Restart : BOOL;
       PRestartAllowedDuringInterlock : BOOL;
END_VAR
VAR_OUTPUT
       out_InterlockNotAcknowledged : BOOL;
       out_AlarmUnacknowledged : BOOL;
       out_RestartAllowed : BOOL;
END_VAR;
    IF in_Acknowledge THEN
        out_InterlockNotAcknowledged := FALSE;
        out_AlarmUnacknowledged := FALSE;
    ELSIF in_Interlock THEN
        out_AlarmUnacknowledged := TRUE;
    END_IF;
    IF (in_Restart AND NOT in_Interlock) OR
       (PRestartAllowedDuringInterlock AND in_Restart AND
            in_Interlock)
       AND NOT out_InterlockNotAcknowledged THEN
        out_RestartAllowed := TRUE;
    END_IF;
    IF in_Interlock THEN
        out_InterlockNotAcknowledged := TRUE;
        out_RestartAllowed := FALSE;
    END_IF;
END_FUNCTION_BLOCK
```

Listing 1: Example PLC code.

Table 1: Counterexample for the Example Requirement

Variable	Cycle 1	Cycle 2
in_Acknowledge	FALSE	FALSE
in_Interlock	TRUE	FALSE
in_Restart	FALSE	TRUE
PRestartAllowedDuringInterlock	FALSE	FALSE
out_InterlockNotAcknowledged	TRUE	*TRUE*
out_RestartAllowed	FALSE	*TRUE*

Even checking this small PLC code can show some of the advantages of model checking. The code above is an excerpt from a real PLC program. We were able to find the same fault using the same requirement on the whole PLC program, without knowing its presence a priori. As the number of inputs in the original program is much higher, it would be extremely difficult to be checked using testing.

CONCLUSION AND FUTURE WORK

We have presented an automated method for model checking PLC programs. This method is incorporated by the tool PLCverif that allows users not expert in formal methods to use model checking without a need for long training. We have applied this method for many PLC programs at CERN and we have found several problems in supposedly well-tested, mature PLC programs, including the example presented above.

Again, model checking is not a solution for all verification-related challenges. The models generated from PLC programs can be huge, even after using reduction techniques. If a requirement can be checked using a couple of test cases, then testing can have an advantage. Nevertheless, model checking can complement testing by providing solutions for different types of requirements, where testing can be inefficient or practically impossible.

Related Work. There are only a few of available tools providing formal verification of PLC code: the most known is Arcade.PLC [8] that focuses on model checking of PLC code. While they provide solutions for hiding the difficulties of building the formal models, the requirements should be provided directly in CTL or LTL. Many other authors propose other solutions, but their tools are not available (not downloadable, nor described in detail in any paper). Another constraint is the support of the ST language which is not common in those tools.

Future Work. The future work is twofold. First, the reductions and model checking algorithms could be improved to increase the set of verifiable problems. The second is coming from the fact that model checking needs unambiguous requirements to be verified. If there is no formal specification for the program, it is difficult to extract these requirements. Therefore we are working on the design of formal specification methods for the PLC domain [9].

REFERENCES

[1] B. Fernández, E. Blanco, and A. Merezhin, "Testing & verification of PLC code for process control," in *Proc. of the 14th Int. Conf. on Accelerator & Large Experimental Physics Control Systems*, 2013, pp. 1258–1261.

[2] B. Fernández, D. Darvas, J.-C. Tournier, E. Blanco, and V. M. G. Suárez, "Bringing automated model checking to PLC program development – A CERN case study," in *Proc. of the 12th Int. Workshop on Discrete Event Systems*. IFAC, 2014, pp. 394–399.

[3] D. Darvas, B. Fernández, A. Vörös, T. Bartha, E. Blanco, and V. M. G. Suárez, "Formal verification of complex properties on PLC programs," in *Formal Techniques for Distributed Objects, Components, and Systems*, ser. LNCS. Springer, 2014, vol. 8461, pp. 284–299.

[4] *IEC 61131-3:2013 Programmable controllers—Part 3: Programming languages*, IEC Std., 2013.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proc. of the 21st Int. Conf. on Software Engineering*. ACM, 1999, pp. 411–420.

[6] J. C. Campos, J. Machado, and E. Seabra, "Property patterns for the formal verification of automated production systems," in *Proc. of the 17th IFAC World Congress*. IFAC, 2008, pp. 5107–5112.

[7] E. Blanco *et al.*, "UNICOS evolution: CPC version 6," in *Proc. of the 13th Int. Conf. on Accelerator & Large Experimental Physics Control Systems*, 2011, pp. 786–789.

[8] S. Biallas, J. Brauer, and S. Kowalewski, "Arcade.PLC: A verification platform for programmable logic controllers," in *Proc. of the 27th IEEE/ACM Int. Conf. on Automated Software Engineering*. ACM, 2012, pp. 338–341.

[9] D. Darvas, E. Blanco, and I. Majzik, "A formal specification method for PLC-based applications," in *Proc. of the 15th Int. Conf. on Accelerator & Large Experimental Physics Control Systems*, 2015, in press.