Formal Verification of Safety PLC Based Control Software

Dániel Darvas[1,2(✉)], István Majzik[2], and Enrique Blanco Viñuela[1]

[1] CERN – European Organization for Nuclear Research, Geneva, Switzerland
{ddarvas,eblanco}@cern.ch
[2] Budapest University of Technology and Economics, Budapest, Hungary
{darvas,majzik}@mit.bme.hu

Abstract. Programmable Logic Controllers (PLCs) are widely used in the industry for various industrial automation tasks. Besides non-safety applications, the usage of PLCs became accepted in safety-critical installations, where the cost of failure is high. In these cases the used hardware is special (so-called fail-safe or safety PLCs), but also the software needs special considerations. Formal verification is a method that can help to develop high-quality software for critical tasks. However, such method should be adapted to the special needs of the safety PLCs, that are often particular compared to the normal PLC development domain. In this paper we propose two complementary solutions for the formal verification of safety-critical PLC programs based on model checking and equivalence checking using formal specification. Furthermore, a case study is presented, demonstrating our approach.

Keywords: PLC · Model checking · Formal specification · Safety-critical systems

1 Introduction and Motivation

Programmable Logic Controllers (PLCs) are special industrial computers, widely-used for various automation tasks. Although initially PLCs were not specifically targeting safety-critical applications, it is feasible and increasingly accepted to use these controllers in critical settings with some restrictions [9].

Most of the PLCs can be programmed in one of the languages defined by the IEC 61131-3 standard: Instruction List (IL), Structured Text (ST), Ladder Diagram (LD), Function Block Diagram (FBD) and Sequential Function Chart (SFC). The first two languages are textual with different levels of abstraction: ST is a high-level language, while IL is "assembly-like". The last three languages are graphical. As SFC is a special-purpose language for structuring the PLC programs, this paper focuses on the first four languages. Short examples of these four languages can be seen in Fig. 2. These example programs have the same meaning and behaviour, i.e. they provide the same output sequences for the same input sequences.

E. Ábrahám and M. Huisman (Eds.): IFM 2016, LNCS 9681, pp. 508–522, 2016.
DOI: 10.1007/978-3-319-33693-0_32

Safety-Critical PLC-Based Systems. The safety-critical controllers have to fulfil the requirements of the corresponding standards, such as IEC 61508, IEC 61511, or IEC 62061. These standards define different safety integrity levels (SIL) and various requirements and guidelines for the system and the development process. Many PLC vendors produce a special range of hardware complying with the corresponding standards. These so-called fail-safe PLC CPUs (or simply *safety PLCs* in the following) are typically certified up to SIL3 according to IEC 61508-2. Besides the special hardware, the PLC vendors provide special development environments, often with additional restrictions compared to the non-safety-critical PLC programming. For instance, Siemens[1] restricts the developer to use the LD or FBD language with further restrictions, such as no floating-point or compound data types can be used [17], following the recommendations of the IEC 61511-2 standard. Although the hardware of the safety PLCs is special, the hardware differences do not affect the software part. Thus the main particularity of the safety PLCs for us is the restricted programming possibilities, namely the obligation to use restricted LD or FBD language for programming.

Typically, testing is applied to assess and improve the quality of PLC-based applications. In safety-critical settings more precise verification is needed. Formal verification is not widely used yet in the industry, presumably because of its high cost and complexity. However, as the cost of failure is high in safety-critical PLC-based systems, they are good candidates for formal verification. Fortunately, formal verification becomes more and more accessible thanks to the new methods that hide the difficulties from the developers.

The goal of this work is to apply formal verification (model checking and equivalence checking) to safety-critical PLC programs in order to complement the current verification methods, to increase the quality of the programs by finding more faults. However, the goal is not (yet) to prove the correctness of the PLC programs, therefore we will not focus directly on proving the correctness of our methods in this paper. We extend the PLCverif approach [7], already providing a scalable and flexible model checking method adapted to PLC programs, to make it suitable for the verification of safety-critical systems. This involves three main tasks: (1) support for the specific languages used in safety PLCs, (2) development of new reduction heuristics to cope with large safety programs, and (3) introduction of a new verification approach based on complete behaviour specification. Furthermore, we present a case study, where our method proved to be applicable and useful.

Case Study. CERN, the European Organization for Nuclear Research operates a particle accelerator complex, comprising the Large Hadron Collider (LHC).

[1] As Siemens is widely-used at our organization, we are using it as an example PLC provider. The languages used in Siemens PLCs are compliant with the IEC 61131 standard, but small syntactic and semantics differences exist. The Siemens variants have different names: instead of IL, ST, LD, FBD, SFC, they are called STL, SCL, LAD, FBD, SFC/GRAPH, respectively. To avoid the confusion, we will use the standard language names for the Siemens variants too, but when a detail is vendor-specific, we will use the Siemens syntax or implementation.

Fig. 1. SM18 Cryogenic Test Facility (©CERN, 2013. CERN-GE-1304099-24)

The high collision energy of the LHC necessitates a strong magnetic field to bend the particle beams, achieved by superconducting magnets. These magnets should be tested before putting them into production. For this, CERN has a unique testing facility (so-called *SM18 Cryogenic Test Facility*) where the magnets can be tested at low temperature (1.8 K, achieved by liquid helium and nitrogen), high currents (14 kA) and vacuum. A photo of the SM18 test hall can be seen in Fig. 1 (4 out of the total 10 test benches are shown in the photo, with a white, shorter quadrupole and a blue, longer dipole magnet currently under test). Testing the magnets is a safety-critical task, as a failure can cause serious damage or injury. Therefore a safety instrumented system is in use to allow or forbid the magnet tests based on whether their preconditions are met. Recently a project started to re-engineer this safety system based on safety PLCs. In this project we have applied formal methods from the beginning of the development.

Structure of the Paper. Section 2 introduces the original PLCverif verification approach, that is not adapted yet to safety-critical PLC programs. Then Sect. 3 defines two extensions, making the method applicable in safety-critical settings. The validation and our experiences on the above-presented case study are discussed in Sect. 4. Section 5 presents the related work on formal verification of PLC programs. Section 6 summarizes and concludes the paper.

2 The Original PLCverif Approach

The PLCverif tool[2] provides a scalable and flexible workflow for the model checking of PLCs [5,7]. It has already proven to be useful for non-safety-critical programs, written in ST language [7]. However, as it does not support the FBD and LD languages, PLCverif cannot be used as it is for the verification of our safety-critical PLC programs.

The original workflow (see Fig. 3) builds on two inputs: (1) the ST source code (created by an engineer) and (2) the requirements formalized using pre-defined

[2] http://cern.ch/plcverif/

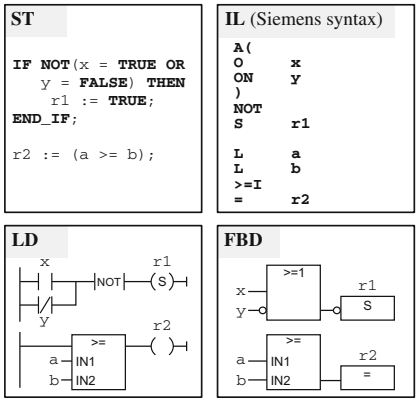


Fig. 2. PLC language examples

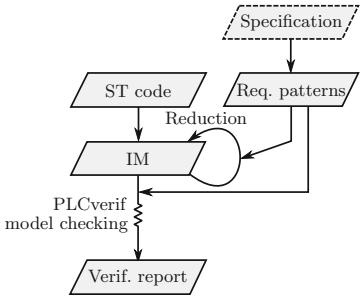


Fig. 3. Original verification workflow

requirement patterns. A requirement pattern is an English sentence containing gaps to be filled by the user with simple expressions. The meaning of the sentence is formalized using temporal logic, having the same placeholders.

First, the ST source code is parsed and translated to an internal, automata-based *intermediate model* (IM). After, based on the given requirement and generic PLC knowledge, the IM is reduced, preserving the properties to be checked [6]. Then the "PLCverif model checking" step is performed: (a) the reduced IM and the requirement are translated to one of the supported model checker's input format; next (b) the model checker tool is executed; and finally (c) the output of the model checker tool is parsed, analysed and presented to the user in a verification report. At the moment the concrete syntaxes of NuSMV, nuXmv, UPPAAL, BIP and ITS tools are supported.

This method has three main advantages:

- **Scalability.** The automated reductions make the verification of large programs possible.
- **Flexibility.** The usage of IM allows to exchange the model checker tools.
- **Usability.** No special knowledge about formal verification is needed from the user: the input of the PLCverif tool is an ST source code and a filled requirement pattern, and the output is a self-contained verification report.

The PLCverif approach was found to be practical and applicable in real cases [7]. However, to reuse this workflow for the safety-critical PLC programs of CERN, three main extensions are needed.

- **Support for new languages.** The (Siemens) safety PLCs can only be programmed in LD or FBD languages, therefore these languages should be supported by PLCverif.
- **Sustain the scalability.** The newly targeted languages are on a lower abstraction level than the ST language, therefore new, specialized reduction heuristics are needed to cope with verification of the large PLC programs.

– **Detailed behaviour checking.** The original PLCverif approach is based on requirement patterns, thus on temporal logic expressions. This is convenient to express some state reachability problems or general safety requirements in a declarative way. However, it is difficult to cover all behaviours with requirement patterns. Besides these requirements, checking the detailed (step-by-step) behaviour is also important for the safety-critical applications. Therefore a complementary method, built on *behaviour equivalence checking* between the implementation and a formal specification, is more convenient to capture the detailed behaviour of the implementation.

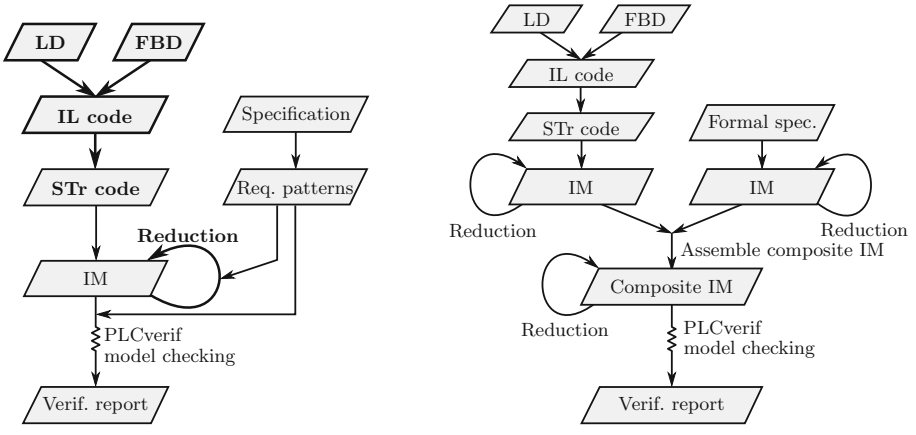
The details of these extensions are discussed in the next section.

3 Extended Approach for Programs of Safety PLCs

This section is dedicated to the extensions of the PLCverif workflow that are necessary to use it for safety-critical PLC programs. Section 3.1 discusses the extensions required to handle the LD and FBD languages. Section 3.2 presents a complementary workflow, built on formal specification and equivalence checking.

3.1 Verification of LD and FBD Programs

The primary need to verify safety-critical PLC programs is the ability to check LD and FBD codes. However, in case of Siemens PLCs, the programs written in graphical languages are not directly accessible, but they can be exported from the development environment as IL code. This solves the problem of parsing LD and FBD languages. However, the abstraction level of IL is even lower than LD's or FBD's, thus it is more difficult to handle IL programs. Our extended workflow can be seen in Fig. 4a, where the new parts are denoted by bold letters.



(a) Workflow to check requirements on safety PLC programs

(b) Workflow to check conformance of safety PLC programs

Fig. 4. Extended verification workflows

Handling IL Code Inputs. The ST parser of PLCverif is based on Xtext[3], providing rich tooling for the language defined by a grammar. However, IL cannot be conveniently represented using Xtext. For example, in Siemens IL "`A`" may be a variable and an AND logic instruction in the same program.

It is resource demanding to implement (1) a parser that can build the abstract syntax tree of the IL language, and (2) a model translator that translates the syntax tree to the intermediate model. Instead of developing these, we have decided to represent IL code as ST code, providing a mapping in an inductive way from each IL instruction to ST instructions. This way the PLCverif model translator does not change, also the instruction-by-instruction mapping can be much simpler than a complete parser in case of the IL language.

The challenge of this mapping is that the IL instructions directly access and modify the different registers[4] of the PLC. For example, the instruction "`L var1`" stores the contents of Accumulator 1 in Accumulator 2, then it loads the value of variable `var1` to Accumulator 1. There is no language element to access the registers directly in ST, making the direct representation of IL code impossible. However, this can easily be solved for verification purposes. We emulate the registers as local ST variables according to a well-defined naming convention, and use it consistently in the ST programs and in the properties to be verified. To avoid the confusion – though it does not require a language extension –, we will use *STr* as language name for programs written in ST where the registers are emulated as local variables. This solution is similar to the one presented in [19]. To distinguish between ordinary variables and the ones representing STr registers, the latter's names start with double underscores.

With this extension, each IL instruction (e.g. bit logic and comparison operations, conversions, jumps, arithmetic instructions, load and transfer instructions) can be represented in STr, by making all implicit effects of the IL instructions explicit in STr. For this purpose, we have identified the semantics of each IL instruction by checking on real PLCs what are the results of the instruction for every possible initial state (i.e. for each valuations of the read registers and variables). The identified semantics of the IL instructions are generic, not specific to our case studies. Some examples for this translation with different complexities are in Table 1. A short description of the used registers is in Table 2[5].

As each IL instruction can be translated into STr, it can be seen inductively that each IL program can be translated into STr as well. In other words, STr can emulate all IL programs, and consequently all FBD and LD programs too. Furthermore, STr can be regarded as a textual concrete syntax of the PLCverif intermediate model (IM), therefore there is no theoretical difference if we translate IL programs to the intermediate model directly or through STr.

[3] https://www.eclipse.org/Xtext/

[4] As previously discussed, we use the Siemens notations in this paper. Throughout this paper *registers* are used as a generic term referring to the status bits, accumulators and the nesting stack.

[5] Here we omit the registers not necessary for simple IL programs, such as the BR (binary result), OV (overflow), OS (stored overflow) bits and the address registers.

Table 1. IL to STr transformation examples

IL	STr equivalent
`A var1`	`IF __NFC THEN __RLO:=__RLO AND (var1 OR __OR); ELSE __RLO:=var1 OR __OR; END_IF;` `__STA:=var1; __NFC:=TRUE;`

`A(`	`__nsRLO[8]:=__nsRLO[7]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO OR NOT __NFC;`
	`__nsOR[8] := __nsOR[7]; ... __nsOR[2] := __nsOR[1]; __nsOR[1]:=__OR AND __NFC;`
	`__nsFC2[8]:=__nsFC2[7]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=FALSE;`
	`__nsFC1[8]:=__nsFC1[7]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=FALSE;`
	`__nsFC0[8]:=__nsFC0[7]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=FALSE;`
	`__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;`

`>I`	`__OR:=FALSE; __NFC:=TRUE;` `__RLO:=(__ACCU1<__ACCU2); __CC0:=(__ACCU1>__ACCU2); __CC1:=(__ACCU1<__ACCU2);`
`L var1`	`__ACCU2 := __ACCU1; __ACCU1 := var1;`

Table 2. Main registers in Siemens PLCs [16]

Register	Purpose
`__RLO`	*Result of last logic operation.*
`__OR`	Helper bit for the "and before or" logical operation (`O` instruction).
`__NFC`	*Not first computation.* If it is false, the current value of `__RLO` is not taken into account.
`__STA`	*Status bit.* Stores the value of a bit that is referenced.
`__CC0`, `__CC1`	*Condition codes.* The result of the last comparison or other operations.
`__ACCU*`	*Accumulators.*
`__ns*[]`	*Nesting stack.* Temporarily stores register values (`__nsRLO`, `__nsOR`) and the last Boolean operation (`__nsFC*`) while a nested Boolean computation is in progress.

Code Size Blow-Up and Reductions. Representing the registers as local variables allows the inductive mapping of IL programs to the ST language, making possible to reuse the PLCverif workflow and toolchain. However, it raises a new concern: a single IL instruction may read and modify several registers. This causes a significant blow-up, as illustrated in Fig. 5. The original sample IL code contains 4 instructions (Fig. 5a), that can be represented by one single statement in ST (Fig. 5b). However, the IL code translated to STr have 14 variable assignments (Fig. 5c). Note that these assignments represent the storage of (intermediate) results that are not necessarily needed by the subsequent statements. The extremities of this are the nesting Boolean operators (e.g. "`A(`"). They store some intermediate computation results in the so-called nesting stack, therefore a single IL operation might be translated to 40–50 STr assignments (see Table 1).

This blow-up effect can be reduced by developing new automated reduction heuristics, similarly to the ones already included in the PLCverif workflow [6]. The new reductions are similar to the reductions used in optimized compilers.

– *Expression propagation* can help to reduce the number of assignments. For example, the second assignment of line 1 in Fig. 5c can be removed and the

```
1  L a
2  L b
3  >=I
4  = r
```

(a) Source IL code (Siemens)

```
1  r := (a >= b);
```

(b) Equivalent ST code

```
1  __ACCU2 := __ACCU1;  __ACCU1 := a;
2
3  __ACCU2 := __ACCU1;  __ACCU1 := b;
4
5  __OR := FALSE;  __NFC := TRUE;
6  __RLO := (__ACCU1 <= __ACCU2);
7  __CC0 := (__ACCU1  > __ACCU2);
8  __CC1 := (__ACCU1  < __ACCU2);
9
10 IF __MCR THEN r := __RLO; END_IF;
11 __OR := FALSE;  __STA := r;  __NFC := FALSE;
```

(c) Generated STr code

Fig. 5. Illustration of code blow-up caused by IL to STr translation

first assignment of line 3 can be replaced by `__ACCU2 := a;` without modifying the behaviour of the program.

- The *assignments without observable effect* can be removed. For example, the first assignment of line 1 in Fig. 5c can be removed, as its effect is hidden by the first assignment of line 3.
- The non-used variables are deleted by the already existing *cone of influence* reduction. For example, the `__CC0` and `__CC1` variables can be removed, as they are never read in Fig. 5c.
- The expression propagation can result in complex Boolean expressions, that can be reduced by *Boolean factoring* and other *Boolean expression reduction methods*. If the simplified expression refers to fewer variables, these reductions may help the cone of influence reduction. Nevertheless, even if they do not reduce the state space, the Boolean expression simplification makes the other reductions faster and decreases the memory needs.

By using these reduction heuristics, the code in Fig. 5c can be automatically reduced to the one in Fig. 5b, when the registers are not read by any further part of the code. Note that each reduction is applied only if it preserves the properties that are currently under evaluation.

3.2 Verification Based on Formal Specification

The previous approach reused the original pattern-based requirement specification method. This is a suitable way to check state reachability properties expressed by the developer. However, the verification of the detailed behaviour is similarly important in safety-critical systems, for which the original approach is not convenient. Furthermore, there is no guarantee that the verification based on manually extracted requirements covers all important aspects of the code. In extreme cases the verification of these requirements can have an opposite effect: the developer convinces himself based on incomplete requirements that the implementation is correct. Later this might bias the testing process. To avoid this, we provide a complementary verification approach which is more convenient for detailed behaviour verification.

The specification of PLC programs is an important topic, yet there are no widely used behaviour specification methods, especially not formal methods with precise semantics definition. Providing a detailed specification is often too "expensive", and instead of precisely specifying the behaviour, documents written in natural language and informal control tables are used that are easy to misunderstand and difficult to verify. However, as the cost of failure is high in the safety-critical domain, also the behaviour of a safety PLC program is typically simpler. Therefore providing a formal specification may be feasible in these cases.

Previously we have proposed a method called *PLCspecif* [4] for the formal specification for PLC programs. Its aim is to provide a formal, yet convenient way for the PLC developers to describe a detailed, complete behaviour specification of the module or system under development. PLCspecif plugs together different, already used semi-formal description methods, e.g. state machines, data-flow diagrams, truth tables; and assigns a unified formal semantics to them. This helps the development and the verification by providing unambiguous requirements. The semantics of PLCspecif is designed in a way that the specifications can be easily transformed to automaton-based models for formal verification.

If such specification exists for the safety-critical PLC program, we can benefit from it and check directly the equivalence between the implementation and the specification. This workflow is shown in Fig. 4b. The semantics of PLCspecif is given as an automaton construction, that can be directly represented in the IM formalism of PLCverif. Accordingly, two IMs are used in this approach: one to represent the implementation, and another one representing the specification. First, both IMs are reduced independently. In this phase we only use reduction rules that preserve all properties, assuming that they check variable values only at the end of the PLC cycles. Then the two IMs are automatically combined into a composite verification model. This composite verification model is constructed on the basis of the definition of the behavioural equivalence. As the equivalence relation we would like to check requires that for each possible input sequence, the specification and the implementation give the same output sequences, the composite verification model ensures by construction that the two model parts always get the same input values. After, the composite model is reduced again. Then, similarly to the original PLCverif approach, we use one of the supported external model checkers to decide whether the equivalence relation holds, namely, that the corresponding outputs of the two model parts are equal in each step.

4 Validation and Analysis of Applicability

To demonstrate and validate the presented approaches, we recall the SM18 Cryogenic Test Facility's safety controller. The implementation under test in this case is the safety logic of this controller, isolated from the rest of the program that is responsible for the non-safety-critical tasks. The IL code exported from the original LD implementation contains about 9500 instructions. This was translated into approx. 120 000 STr statements, already with some optimizations (e.g. the nesting stack depth was reduced to the necessary amount). The potential state

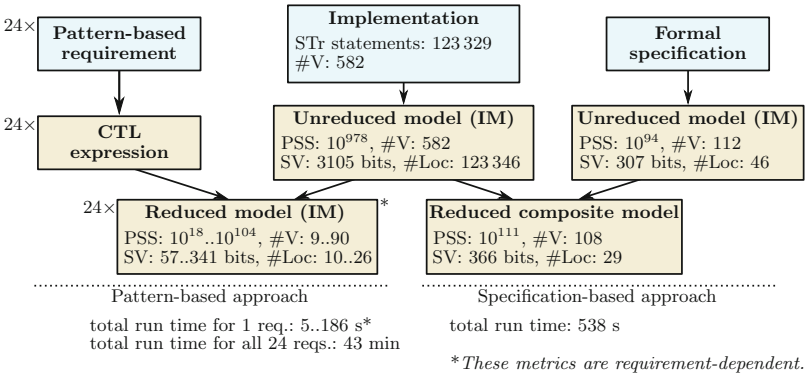


Fig. 6. Key metrics of the example

space (i.e. the cross-product of all contained variable's domain and the possible automata locations in the IM) contains approx. 10^{978} states.

We have applied both previously described formal verification approaches: first the pattern-based, then the specification-based approach. The key metrics and the summary of the two methods can be seen in Fig. 6. For each IM we give the size of potential state space (PSS), the number of variables (#V), the size of the state vector, i.e. the length of binary vector that can represent the current values of all variables (SV), and the number of automata locations (#Loc).

Verification Based on Requirement Patterns. After the successful representation of the safety logic in STr language, we have captured pattern-based requirements from the informal specification provided by the client of the project. As this was the first safety-critical PLC program verification project at CERN, the requirements were extracted by formal methods experts, rather than the PLC program developers. In total 24 different requirements were extracted and formalised using requirement patterns. Some of them are fairly simple, while some others contain references to up to 50 different variables.

In each case the verification was successfully executed, thanks to the requirement-specific and general reductions that reduced both the number of variables and automaton locations. Also, the reductions were able to eliminate all register-representing variables in every case. The typical verification run time of each requirement was 150–170 s, including the model generation, the model reductions and the execution of the external model checker (nuXmv in this case)[6]. In case of some requirements, only a small part of the model was enough for the verification of the given requirement, therefore the reductions were able to eliminate a large part of the IM, resulting a total run time of 4–5 s. The total run time of all verification cases together was 43 min. The peak memory consumption of PLCverif was 2926 MB, however as the implementation is in Java,

[6] For all measurements we have used PLCverif 2.0.2 and nuXmv 1.0.1 on Windows 7 x64, executed on a PC with Intel® Core™ i7-3770 3.4 GHz CPU and 8 GB RAM.

this number is an upper estimation of the required memory. The peak memory consumption of nuXmv was 570 MB. In these cases the reductions performed the significant part of the verification, the external model checker was easily able to cope with the reduced model. Even the longest nuXmv execution time was shorter than 30 s, and in many cases it was less than a second. However, without our reductions the model checking could not be possible at all.

As the total run time, as well as the computation resource requirements are significant, an automated solution was built using Jenkins[7], that automatically executes the verification of all requirements on any code or requirement modification in the version control system. The execution takes place on remote servers, this is completely transparent for the user. When the verification is completed, the responsible people are notified by e-mail about the results of the verification.

Verification Based on Formal Specification. To validate the second approach, the formal specification of the magnet test safety logic had to be captured. We did not have a formal specification a priori, the implementation was developed based on a semi-formal specification. As the precise semantics of the client's specification was already clarified during the previous verification process, the creation of the formal specification was relatively simple. Note that PLCspecif provides various tabular description methods, similar to the one used by the customer of the project. As previously discussed, first the PLCspecif specification was automatically translated to intermediate model. Even before reductions, this IM generated from the specification was much smaller than the IM generated from the STr code, as the model size blow-up caused by the explicit representation of PLC registers does not occur in this case. After the reductions, the composite verification model was constructed and reduced. The resulting verification model was larger than the biggest individual verification model generated using the pattern-based approach. Consequently the total run time was longer, approximately 10 min. However, this had to be done only once, while the first approach necessitated 24 verification runs, one for each requirement. Therefore in total the run time of the second approach was more than four times shorter.

Analysis of the Results. After performing the case study we have concluded that the verification was successful, as it was possible to model and verify the critical part of the PLC program. We have applied an iterative workflow: every time the model checking pointed out a problem, we have suspended the verification process until the root cause of the problem was fixed. Then the verification process restarted with the new code version. In total 14 issues were identified. We have classified the problems found into the following main categories:

- 4 *requirement misunderstanding* problems. In these cases the formalization of the requirements pointed out ambiguous or contradictory elements in the specification provided by the customer, overlooked during implementation.
- 3 *functionality* problems. In these cases the problem could have caused unexpected behaviours, but not dangerous situations.

[7] https://jenkins-ci.org/

- 5 *safety* problems. In these cases the problem could have caused dangerous situations, i.e. a magnet test might be permitted when it should not.
- 2 *mixed functionality-safety* problems.

All these problems were found before on-site testing of the PLC program. As the (re-)deployment and the PLC's on-site testing is a time-consuming operation, model checking provided an efficient verification method. Furthermore, model checking does not involve the use of real hardware, therefore no dangerous situations can happen contrarily to on-site testing.

As testing in lab and on site provides the state-of-the-practice in the verification of PLC-based systems, we have checked whether the problems identified using formal methods could have been found using the typically applied testing methods. Setting up a test scenario on-site can take up to hours, therefore only the main functionalities and their most critical errors are targeted, potentially omitting problems. Out of the 10 functionality or safety issues, 4 could have been found using testing. In 6 cases it was practically impossible to find the problem using our regular testing approach, as the testing is not exhaustive in practice.

We have performed the pattern-based verification approach first, which identified 12 of the 14 issues. The remaining 2 problems were found using the specification- and equivalence checking-based approach. This shows that the two methods are convenient for different types of requirements, and they can complement each other. The system is now in production for 7 months. So far no major problem was observed in operation caused by mistakes in the safety logic.

Comparison of the Two Approaches. The two presented verification approaches (pattern-based and behaviour specification-based approaches) provide different advantages and disadvantages.

- Using the behaviour specification-based approach, all requirements contained in the original specification are covered, there is no potential user omission in extracting the requirements.
- The pattern-based approach can check properties in a more "declarative" way, i.e. without specifying the complete behaviour and may help the user to find discrepancies between the general expectations and the implementation, or various requirement misunderstandings.
- The verification models generated in the pattern-based approach are often smaller than in the other approach, providing better verification performance. On the other hand, this approach involves multiple verification runs, whereas the specification-based approach needs only one model generation and one model checker execution.
- The integration to the existing development processes is easier in case of the pattern-based approach, as there is no need for a complete formal specification, which does not exist typically and often difficult to be constructed.

5 Related Work

The formal verification, especially model checking of PLC programs was deeply studied in the last fifteen years [13]. Several approaches were developed with

Table 3. Related work

Ref.	Lang.	Real-life applicability	Scalability	Tool	Verifier tool
[2]	IL, ST	●●	●●	+	Arcade
[3]	IL	●	●	−	CaSMV
[8]	ST, ...	●●	●●	−	NuSMV
[10,20]	FBD	●●	●●●	+	CaSMV
[11]	IL	●●	●●	−	Z3
[12]	SFC	●●	●●	+	SpaceEx
[14]	FBD	●●	●●	−	NuSMV
[15]	LD	●	?	−	UPPAAL
[18]	FBD	●	?	−	UPPAAL
[19]	IL	●	●	−	MiniSat
[7]	ST, SFC	●●●	●●●	+	nuXmv, UPPAAL, ...

Legend ●●●: *high*, ●●: *medium*, ●: *low*; +: *exists*, −: *does not exist*

different advantages and capabilities. We summarize the works most relevant to us in Table 3. To investigate the applicability of the methods, we have four main factors to take into account: the set of supported languages, the real-life applicability (how feasible it is to include the method in the normal PLC development workflow), the scalability of the method and whether a supporting tool was developed. It should be noted that the tools are typically not publicly available, except for [2].

For checking applicability, we used the PLCverif approach [7] as a base of comparison. This comparison contains subjective elements, but we claim that PLCverif provides a better real-life applicability than the other methods, as the formal verification-related difficulties are hidden from the user, there is no need to edit directly temporal logic expressions or to invoke model checker tools. The scalability is a similarly important question. We have tried to judge the scalability of each method based on the cited papers. If it was not possible (e.g. there was no presented verification example), we put "?" in the table. We have also included the used verifier tools in the table. It can be seen that [7] was the only approach we have found that provides a generic approach relying on multiple model checkers, depending on the current verification needs.

Since [7] does not provide support for the FBD and LD languages necessary to verify programs of safety PLCs, we decided to extend this method and to benefit from its advantages in the other dimensions.

[19] is a particularly interesting related work. Their approach is similar to ours: they translate the IL code instruction by instruction into a pivot language, that is SystemC in their case. The verification is performed as an equivalence checking between the SystemC representation of the implementation and the specification. However, the scalability of this method is not justified.

Furthermore, the SystemC specification cannot be used directly in our PLC development workflow due to the lack of specific knowledge.

Equivalence checking was already used in different verification settings for PLC programs: [1] applies regression verification between two versions of the implementation. In our work we apply equivalence checking between the formal behaviour specification and the implementation.

6 Summary and Conclusion

In this paper we presented an extension to the PLCverif approach [7] to handle the PLC programs written in FBD, LD or IL language that is necessary to verify safety-critical PLC programs. As [7] already contains methods to handle the ST and SFC languages, the current paper justifies also the claim that PLCverif can be a generic approach handling all five common PLC languages. To cope with the safety-related languages, additional reduction heuristics were introduced. Besides the requirement pattern and model checking-based verification approach, a new approach was drawn up, based on a PLCspecif formal specification and equivalence checking. These two approaches can complement each other.

A case study was presented showing that formal verification can be applied to significantly large, real safety-critical PLC programs. The two formal verification techniques identified several problems and they complemented each other. Many of the problems identified using them could have not been found using the currently used testing techniques. Moreover, the presented approaches helped to identify problems with the requirements, such as ambiguity or contradictions, overlooked by the developers during implementation. Formal verification was applied in the design phase, thus fixing the problems was easier than if they would have been found during on-site testing or in production. Furthermore, model checking and behavioural equivalence checking provided a safe way to check requirements, without any safety risks that might arise during on-site testing.

Acknowledgement. The authors would like to thank the people involved in the presented re-engineering project for their support and cooperation. Special thanks to Roberto Speroni for the cooperation and the continuous feedback.

References

1. Beckert, B., Ulbrich, M., Vogel-Heuser, B., Weigl, A.: Regression verification for programmable logic controller software. In: Butler, M., Conchon, M., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 234–251. Springer, Heidelberg (2015)
2. Biallas, S., Brauer, J., Kowalewski, S.: Arcade.PLC: a verification platform for programmable logic controllers. In: Proceedings of 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 338–341. ACM (2012)
3. Canet, G., Couffin, S., Lesage, J.J., Petit, A., Schnoebelen, P.: Towards the automatic verification of PLC programs written in instruction list. In: Proceedings of IEEE International Conference on Systems, Man, and Cybernetics, vol. 4, pp. 2449–2454. IEEE (2000)

4. Darvas, D., Blanco Viñuela, E., Majzik, I.: A formal specification method for PLC-based applications. In: Proceedings of 15th International Conference on Accelerator & Large Experimental Physics Control Systems, pp. 907–910. JaCoW, Geneva (2015, in press)
5. Darvas, D., Fernández Adiego, B., Blanco Viñuela, E.: PLCverif: a tool to verify PLC programs based on model checking techniques. In: Proceedings of 15th International Conference on Accelerator & Large Experimental Physics Control Systems, pp. 911–914. JaCoW, Geneva (2015, in press)
6. Darvas, D., Fernández Adiego, B., Vörös, A., Bartha, T., Blanco Viñuela, E., González Suárez, V.M.: Formal verification of complex properties on PLC programs. In: Ábrahám, E., Palamidessi, C. (eds.) FORTE 2014. LNCS, vol. 8461, pp. 284–299. Springer, Heidelberg (2014)
7. Fernández Adiego, B., Darvas, D., Blanco Viñuela, E., Tournier, J.C., Bliudze, S., Blech, J.O., González Suárez, V.M.: Applying model checking to industrial-sized PLC programs. IEEE. Trans. Ind. Informat. **11**(6), 1400–1410 (2015)
8. Gourcuff, V., de Smet, O., Faure, J.M.: Improving large-sized PLC programs verification using abstractions. In: Proceedings of the 17th IFAC World Congress, pp. 5101–5106. IFAC (2008)
9. Greenway, A.: A user's perspective of programmable logic controllers (PLCs) in safety-related applications. In: Redmill, F., Anderson, T. (eds.) Technology and Assessment of Safety-Critical Systems, pp. 1–20. Springer, London (1994)
10. Jee, E., et al.: FBDVerifier: interactive and visual analysis of counterexample in formal verification of function block diagram. J. Res. Pract. Inf. Technol. **42**(3), 171–188 (2010)
11. Lange, T., Neuhäußer, M.R., Noll, T.: Speeding up the safety verification of programmable logic controller code. In: Bertacco, V., Legay, A. (eds.) HVC 2013. LNCS, vol. 8244, pp. 44–60. Springer, Heidelberg (2013)
12. Nellen, J., Ábrahám, E., Wolters, B.: A CEGAR tool for the reachability analysis of PLC-controlled plants using hybrid automata. In: Bouabana-Tebibel, T., Rubin, S.H. (eds.) Formalisms for Reuse and Systems Integration. AISC, vol. 346, pp. 55–78. Springer, Heidelberg (2015)
13. Ovatman, T., Aral, A., Polat, D., Ünver, A.O.: An overview of model checking practices on verification of PLC software. Software & Systems Modeling, 1–24 (2014). doi:10.1007/s10270-014-0448-7. Advance online publication
14. Pavlović, O., Ehrich, H.D.: Model checking PLC software written in function block diagram. In: Proceedings of International Conference on Software Testing, Verification and Validation, pp. 439–448. IEEE (2010)
15. Sarmento, C.A., Silva, J.R., Miyagi, P.E., Santos Filho, D.J.: Modeling of programs and its verification for programmable logic controllers. In: Proceedings of the 17th IFAC World Congress, pp. 10546–10551. IFAC (2008)
16. Siemens: Statement List (STL) for S7–300/S7-400, C79000–G7076-C565-01 (1998)
17. Siemens: SIMATIC Industrial Software SIMATIC safety – Configuring and Programming, A5E02714440-AD (2014)
18. Soliman, D., Frey, G.: Verification and validation of safety applications based on PLCopen safety function blocks. Control Eng. Pract. **19**(9), 929–946 (2011)
19. Sülflow, A., Drechsler, R.: Verification of PLC programs using formal proof techniques. In: Formal Methods for Automation and Safety in Railway and Automotive Systems, pp. 43–50. L'Harmattan, Budapest (2008)
20. Yoo, J., Cha, S., Jee, E.: A verification framework for FBD based software in nuclear power plants. In: Proceedings of the 15th Asia-Pacific Software Engineering Conference, pp. 385–392. IEEE (2008)