ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA159

WHAT IS SPECIAL ABOUT PLC SOFTWARE MODEL CHECKING?

D. Darvas*, E. Blanco Viñuela, CERN, Geneva, Switzerland
I. Majzik, Budapest University of Technology and Economics (BME), Budapest, Hungary

Abstract

Model checking is a formal verification technique to check given properties of models, designs or programs with mathematical precision. Due to its high knowledge and resource demand, the use of model checking is restricted mainly to core parts of highly critical systems. However, we and many other authors have argued that automated model checking of PLC programs is feasible and beneficial in practice. In this paper we aim to explain why model checking is applicable to PLC programs even though its use for software in general is too difficult. We present an overview of the particularities of PLC programs which influence the feasibility and complexity of their model checking. Furthermore, we list the main challenges in this domain and the solutions proposed in previous works.

INTRODUCTION AND MOTIVATION

The promise of model checking is to provide precise, mathematically sound means to check the satisfaction of given requirements on models, representing for example software. Although some tools are available (e.g. CBMC[1] [1], BLAST[2], Bandera[3] [2], DIVINE[4] [3]), it is still difficult to use model checking on real-sized software in practice. One of the bottlenecks is the verification performance, the excessive need of resources for the successful verification.

Besides checking software written in general-purpose programming languages (e.g. C, C++, Java), active research can be observed focusing on PLC (Programmable Logic Controller) programs specifically. It has been studied by dozens of research groups over the last 20 years [4]. However, model checking is still far away from being easy-to-use or part of the state of the practice of PLC program development.

The reader may ask the question: what is the reason for targeting PLC model checking specifically? What makes this domain special and why there is a need for specific tools? What makes PLC model checking different from verifying general-purpose programs? This paper is dedicated to the specificities of PLC programs, which facilitate their verification, or contrarily, make the model checking more difficult. Our aim is to summarise our experience with PLC software model checking that we have acquired during the development of PLCverif [5], and to help formal verification researchers to specialise in this field, or to make their model checker tools applicable to the PLC program verification domain too.

* Corresponding author. E-mail: ddarvas@cern.ch
[1] http://www.cprover.org/cbmc/
[2] http://cseweb.ucsd.edu/~rjhala/blast.html
[3] http://bandera.projects.cs.ksu.edu/
[4] http://divine.fi.muni.cz/

The paper first overviews the difficulties and advantages arising from the domain specificities. Then the syntactic and semantic particularities of PLC programs are discussed. Finally, the need for environment modelling is mentioned.

An extended version [6] of this paper is also available which contains more details and example programs.

DOMAIN SPECIFICITIES

Many of the differences between the general-purpose and PLC programming languages, but also between the available verification methods originate from the differences in the respective domains. Therefore, we start by overviewing the most important properties and specificities of the PLC domain which influence the formal verification of PLC programs.

Medium Criticality

Except for trivial programs, it is difficult to imagine and prove absolute correctness or safety, just as absolute security. Instead of pursuing those ideals, a more pragmatic approach is needed: the verification costs and the risks of failure should be in balance. Formal verification is already often used where the cost of failure is exceptionally high: in case of highly critical systems (e.g. nuclear, railway or avionics systems) or systems produced in high quantities (e.g. microprocessors). Even the methods requiring special knowledge and lots of resources may be affordable in those cases. Contrarily, in case of systems with low criticality, deep analysis may not be required.

PLC systems are in the middle of this criticality scale: their criticality is often not high enough to afford an independent, specially skilled verification team. However, a potential failure or outage may cause significant economic losses, motivating a sound and detailed verification approach.

Consequence PLC model checking approaches should be easily accessible, specifically targeting the PLC domain, without requiring unaffordable resources or having an excessive cost compared to the level of criticality.

Advantage: Simple Operations and Data Structure

In general, the functionality of PLC programs is simpler than most programs written in C or Java. PLC programs do not deal with graphical interfaces, large data structures; they do not create files, do not perform complex operations. All these features may complicate the software model checking.

Consequence The simplicity of the programs makes model checking more feasible computationally. This makes the PLC domain a good target for formal verification.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA159

Difficulty: Variety of PLC Languages

PLCs use special languages, not used outside this domain. Furthermore, there is a wide variety of PLC programming languages. IEC 61131, the relevant standard [7] defines five different languages: Structured Text (ST), Instruction List (IL), Function Block Diagram (FBD), Ladder Diagram (LD) and Sequential Function Chart (SFC). Furthermore, each vendor provides their own flavour, with minor or major differences compared to the standard. Siemens PLCs typically support Structured Control Language (SCL), Statement List (STL), Function Block Diagram (FBD), Ladder Logic (LAD), and S7-GRAPH, which correspond to the previously listed standard languages, respectively. The difference between some of them is minor (e.g. between LAD and LD), but in other cases it is very significant (e.g. between STL and IL or SCL and ST).

Consequence As PLC programs can mix these languages (e.g. a function written in FBD can call a function written in SCL), each language should be supported by a PLC program verification tool. Furthermore, as there are common parts in those languages (e.g. variable declarations), the language infrastructure of the verification tool (parser and program representation) should be generic and reusable.

Difficulty: Different Background Knowledge of Developers

General purpose programming languages, their development environments and verification tools are typically developed "inside the community": by software engineers, for software engineers. PLC programs, however, are often written by people with different skills and background knowledge: automation engineers, technicians, etc. The theory and practice of formal verification is often not part of the general curriculum of software engineers, making the application of model checking hard. This knowledge gap is even bigger and more severe in case of the PLC program developers.

Consequence Special attention should be paid to bridge the semantic gap between the user and the verification tool. The tools should use inputs and outputs which are close to the users' domain. For example, the PLCverif tool [5] uses the PLC programs and requirement patterns based on English sentences as inputs, and the outputs are provided in an easy-to-understand, self-contained form, using concepts directly from the PLC domain.

SYNTAX OF PLC LANGUAGES

As mentioned earlier, PLC programs are written using a wide variety of programming languages. Since—according to their claims—Siemens is a market leader in the field of automation, we mainly focus on the languages supported by Siemens S7 PLCs, especially the high-level Structured Control Language (SCL), which is a variant of the Structured Text (ST) language defined in IEC 61131 [7].

In this section, we show that although the PLC programs are simpler, their syntax may actually be more complex than that of general-purpose programs.

Difficulty: Complex Syntax

PLC programming languages—especially their Siemens variants—often have richer and more complex syntax than general-purpose programming languages supported by software model checkers. For example, C (the C99 version) contains 6 basic data types with built-in support[5], Java contains 9, but SCL contains 16 base types, which was extended to 30 in the new version of the language supported by the new development environment (TIA Portal) and the new hardware (e.g. S7-1500).

Consequence Development of the language infrastructure for PLC software model checking needs a lot of effort. As there is no good, reusable language infrastructure available, the entry cost of PLC program verification is high.

Difficulty: No Precise Syntax Definition

The well-established general-purpose languages typically have precisely, often formally defined syntax. For example, the syntax of C is standardized by ANSI, ISO and IEC (ISO/IEC 9899), C# is defined by the ECMA-334 standard. The Java syntax is not standard, but a detailed specification is provided by Oracle. The syntax of standard PLC programming languages are defined in IEC 61131 (with some ambiguities [8, 9]). However, having a precise definition for the vendors' flavours is not always easy. Siemens provided syntax definition for SCL version 5.3 [10] and for STL [11], which cover most, but not all aspects. The authors are not aware of any precise syntax description of the new version of SCL, supported by the new development environment, TIA Portal.

Consequence As the available syntax definitions are partial or too vague, the only way to determine the precise syntax is through systematic trials with the compilers. Creating precise descriptions for the most commonly used PLC programming languages and open source, generic parser implementations could facilitate new researchers to focus on the PLC domain and also to focus the research efforts on the verification challenges.

Difficulty: Absolute and Symbolic Addressing

Each Siemens PLC program contains an editable symbol table, which assigns names ("symbols") to memory locations or program units. This allows to use symbolic addressing, i.e. using names instead of absolute addresses. However it is possible (although considered as bad practice) to mix absolute

[5] Here we not only consider the basic numeric types of C, but also strings. Even though a C string is simply a character array, there is dedicated language-level support for string constants (e.g. "`var = "teststring";`").

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA159

and symbolic addresses. For example, "`var1 := TRUE;`", "`"var1" := TRUE;`"[6] and "`M4.1 := TRUE;`" can have the same meaning if there is a symbol `var1` defined for the memory location `M4.1`.

Consequence In order to support real PLC applications, besides supporting the five languages, the symbol tables shall be supported too. The verification tools should be able to handle the mix of absolute and symbolic addresses, or at least warn the user when an object is referred to using several names.

Difficulty: Permissive Grammars

An additional challenge to be faced when developing the PLC language infrastructure is the permissivity of the grammars. For example, there are at least six syntactic ways to refer to a given bit in the bit memory area: absolute access (e.g. `M4.1`, `%MX4.1`; the `%` and `X` are optional) and indexed access (e.g. `M[4,1]`). Furthermore, symbolic access is also possible, as mentioned before.

Consequence The language infrastructure should have a uniform internal representation to hide these redundant details and simplify the verification task.

Difficulty: Context-dependent Grammar

Another challenge in PLC software model checking arises from the context dependent nature of the programming languages. For example, in the STL language "`A A;`" is a valid statement, where the first "`A`" stands for "AND operation", and the second "`A`" denotes a Boolean variable with name "A".

Consequence These features of the language have to be taken into account when choosing the technology for the language infrastructures. For example, a parser that identifies the keywords first—independently from the context—cannot successfully parse a program written in STL due to the mentioned ambiguities, or certain workarounds are required[7]. It also poses a challenge to provide a single, unified parser for SCL and STL.

SEMANTICS OF PLC LANGUAGES

Not only the syntax of PLC programs is rich, their semantics (i.e. the description how the programs behave during execution) may also impose additional challenges compared to formal verification of general-purpose programs.

PLC Execution Semantics

To provide verification for PLC programs, first the key semantic differences between general-purpose programs and PLC programs have to be understood.

PLC programs are typically executed cyclically. A cycle (so-called *scan cycle* or PLC cycle) consists of (1) sampling the physical inputs (and keeping their values stable in the memory), (2) executing the user code, (3) assigning the computed outputs to the physical outputs. This allows to have consistent input and output signals.

In Siemens PLCs, the scan cycle can be *interrupted*. Cyclic interrupts ensure the periodic execution of a certain piece of code. Diagnostic and error handling interrupts can also be defined. The interrupts and various operating system tasks (e.g. communication) can alter the length of the scan cycle. If the scan cycle exceeds the predefined length, an error-handling block will be executed.

There is a difference in the programming concepts too. Even though the latest IEC 61131 standard introduced object-oriented programming for PLCs, most programs still use functions and function blocks. A *function block* is a stateful function, the values of its variables (except for the temporary variables) are kept even after the execution of the block. The semantics of a function block is similar to a class that has a single member method in object-oriented languages.

Advantage: Simple Memory Handling

The formal verification of PLC programs is greatly facilitated by its simple memory handling. PLC programs use *static typing*: variables are declared explicitly, with a given type. Variables are *strongly typed*: except for some safe cases, explicit type conversions are required between the different data types. However, it has to be noted that SCL permits the use of the special data type `ANY` which can store a reference to any data type.

There is no dynamic memory allocation in PLC programs, all variables and data blocks are allocated statically, at compile-time.

Furthermore, in high-level PLC programming languages (e.g. SCL) pointers are rarely used. In lower-level languages (e.g. STL) pointer usage is sometimes unavoidable. However, even without using pointers explicitly, semantically equivalent constructs may be present. For example, `IB[10]` denotes the value of byte 10 in the input memory area. If the `IB` array is indexed with a variable (`IB[var1]`), then `var1` practically behaves as a pointer.

Consequence Due to simple control structures and the lack of dynamic memory allocation, many popular model checkers, e.g. NuSMV or UPPAAL can efficiently be used to verify most of the PLC programs. To support all PLC programs, pointer support is required on the verification side.

Difficulty: Imprecise Semantics Definition

Having a precise, formal semantics for the input models is an obvious requirement for model checking. Unfortunately, there is no mathematically sound semantics definition neither for the IEC 61131 languages, nor for the Siemens PLC languages. Some reference manuals are available for SCL [10]

[6] The quotation marks denote that `var1` is a symbol, however in SCL v5.3 they can be omitted if it does not cause any confusion.

[7] Such workaround for Xtext is in `https://blogs.itemis.com/en/xtext-hint-identifiers-conflicting-with-keywords`.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA159

and STL [11], but they are in some places ambiguous, imprecise or incorrect. For example, the SCL description does not define precisely the semantics of `CASE` statements, or the STL description incompletely and sometimes incorrectly defines the behaviour of the nesting stack used for complex Boolean operations.

PLC programs depend on a library of basic functions and function blocks, such as timers, data transmission blocks, special memory operations. Precise description (either formal definition or source code) is required for these program units too for the verification, but it is often not available.

Consequence Developers of PLC verification tools cannot fully rely on the provided language descriptions and documentation. Systematic, rigorous experiments have to be conducted in order to explore the precise semantics of the different PLC program structures.

No Short Circuit Evaluation

The IEC 61131 standard permits the short-circuit evaluation for logic expressions, i.e. the evaluation can be interrupted as soon as the result can be determined. However, our experiments showed that Siemens PLC programs do not use short circuit evaluation. For example, in case of the "`func1() OR func2()`" expression the function `func2` will be called even if the return value of `func1` is `true` (thus the expression will be evaluated to `true` independently from `func2`).

Consequence This may facilitate the representation of PLC programs as control flow graphs.

Difficulty: Timed Behaviour

PLC programs often involve time-related behaviour, typically by using the timers defined in [7] (TP, TON, TOF). Accurate modelling and verification requires precise representation of time, which might make the verification task extremely difficult. In reality, PLC timers rely on the PLC's real time representation. The elapsed time between two timer calls depends on the cycle time, which in turn relies on the executed methods, the precise type of the hardware, the communication between the PLC and other systems, etc.

Consequence The verification tool should use an appropriate time representation, i.e. an appropriate trade-off between precision of modelling and needed resources. One possibility is to simplify the physical time handling and assuming that each PLC cycle takes a non-deterministic amount of time, and the global time is incremented by this value at the end of the cycle at once. Then effectively the time does not elapse during a PLC cycle, which may alter the behaviour of the timer blocks, but this was often found to be an acceptable trade-off. This representation may lead to false negatives, i.e. omitted faults. Other time representations could cause false positives (false error reports). The consequences of the chosen time representation shall be clearly described for the user, using the terminology of the PLC domain.

Difficulty: Semantics Depending on the Compiler and Hardware Version

The precise semantics of PLC programs may depend on various compiler settings, the used compiler and the hardware.

- Certain data types and languages are available only using certain hardware.
- The precise semantics of the programming languages depend on the development environment.
  Example. Let `D` be a variable of type `DINT` (32 bit signed integer). Using the STEP 7 V5.5 development environment, the execution of the SCL assignment "`D := INT#1 + 50000`" will result in `D` =50001 (where "`INT#1`" denotes a 16 bit signed integer). However, the same code compiled using the TIA Portal development environment will result in `D` =–15535 on the same hardware due to the differences in typing rules.
- The behaviour depends also on the semantic settings. For example, some details of the SFC execution can be modified.
- The hardware configuration and the interrupt configuration can also influence the precise semantics of a given PLC program. Furthermore, this information is not included in the source code.

Consequence Different semantic variants of PLC languages shall be supported, and the user shall be able to choose the appropriate one for each program under verification.

Difficulty: Bit-level Memory Manipulation

PLC programs allow various low-level memory manipulations.

- Integer variables can also be treated as bit arrays by using explicit type conversion operators. The same behaviour is also possible by defining so-called views, practically declaring multiple variables mapped to the same memory location using the keyword `AT` in SCL.
- It is also possible to directly address a specific area in the memory (absolute addressing), independently from the variable borders. For example, `DB1.DW3` refers to the `WORD` starting at byte 2 in the data block `DB1`. However, this memory location may represent several variables, or parts of different variables.

Consequence The verification tool should either provide accurate, low-level representation of the PLC memory model (causing a high overhead), or at least provide static analysis methods to check situations where such advanced memory representation is required to check the PLC program under verification.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA159

ENVIRONMENT

Challenge: Environment Model

PLCs are mainly used for process control tasks, therefore they inherently interact with their environment. It is reasonable to check certain safety properties (i.e. a given property is always satisfied, no matter what are the input sequences) without considering the environment during verification. However for other types of requirements having no assumption on the environment may lead to many false positives, i.e. non-satisfied requirements where the violation is practically impossible.

Consequence To get practical, usable verification results, the model of the environment needs to be incorporated. This can exclude cases where for example only a physically impossible change in the controlled process could cause the signalled violation. In our opinion, there are three main challenges related to the environment models, as follows.

- It is difficult to find appropriate formalisms and to describe the environment (e.g. the controlled process) precisely.
- Including the environment model may significantly increase the computation resources required for model checking.
- An imprecise environment or process model may lead to false negative results, i.e. it can lead to the omission of real problems.

There are various attempts to precisely describe environment or process models and include them in various verification procedures [12–14], however, we think that this still remains one of the greatest challenge in PLC model checking.

Challenge: Fault Assumptions

It is important to keep in mind that the input variables of the PLC programs often represent physical inputs. It is unrealistic to assume that all inputs are always correct. In other words, the "no failure" assumption in the environment model during verification may hide potential problems. The other extreme—assuming that everything can fail at the same time—may be unrealistic too, leading to useless counterexamples which undermines the usability of the method.

Consequence The environment models shall be able to incorporate various assumptions. For example, a single failure hypothesis may be rational in some cases, but in other cases including the simultaneous failure of certain dependent signals in the verification may be desired too.

OUR RESPONSE: PLCverif

To overcome most of these challenges and to provide feasible, easy-to-use formal verification for PLC programs, CERN started the development of PLCverif [5]. With the ongoing development of PLCverif we aim to provide a generic tool and language infrastructure that can make the development or integration of new verification methods to the PLC domain significantly easier.

PLCverif hides the formal verification-related details from the user. Also, as it relies on a control flow graph-based intermediate representation that is independent from the PLC programming languages, this tool can hide many of the syntactic and semantic peculiarities of the PLC domain from the (formal) verification solutions.

Recognizing that the listed particularities make the development of any verification method challenging for PLC programs, PLCverif is opening towards supporting other verification techniques besides model checking, for example static code analysis and unit testing.

Although we have overcame many syntactic and semantic problems—except the ones which would have required unreasonable amount of resources compared to their pertinence, such as properly supporting pointers—, the lack of proper environment modelling limit the use of PLCverif to well-defined, isolated parts of PLC applications, such as individual function blocks or safety logic implementations.

CONCLUSION

In this paper many of the specific challenges of model checking PLC programs have been presented, as well as the features of those programs which can facilitate their formal verification. We believe that PLC model checking is still a research field with a lot of industrial attention and with many unsolved challenges.

On one hand, PLC program verification is an ideal target for model checking due to the medium criticality and the relatively simple programs.

On the other hand, syntax and semantics of PLC programs are complex, which makes it difficult for non-PLC experts to contribute to verification, as the knowledge and development effort required for PLC program verification is high. Open source, reusable language infrastructures could leverage this challenge, allowing to focus on the challenges of performance and clarity of results. We need a bridge not only between formal verification and the PLC developer community, but also between the formal verification researchers and the industrial control systems domain. Furthermore, environment modelling is still a big challenge to be solved, which could significantly improve the practical applicability of model checking for PLC programs.

REFERENCES

[1] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 2988. Springer, 2004, pp. 168–176.

[2] J. Hatcliff and M. Dwyer, "Using the Bandera tool set to model-check properties of concurrent Java software," in *CONCUR 2001 — Concurrency Theory*, ser. LNCS. Springer, 2001, vol. 2154, pp. 39–58.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA159

[3] J. Barnat *et al.*, "DiVinE 3.0 – An explicit-state model checker for multithreaded C & C++ programs," in *Computer Aided Verification*, ser. LNCS. Springer, 2013, vol. 8044, pp. 863–868.

[4] T. Ovatman, A. Aral, D. Polat, and A. O. Ünver, "An overview of model checking practices on verification of PLC software," *Softw. Sys. Modeling*, vol. 15, no. 4, pp. 937–960, 2016.

[5] D. Darvas, B. Fernández, and E. Blanco, "PLCverif: A tool to verify PLC programs based on model checking techniques," in *Proc. 15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems*. JACoW, 2015, pp. 911–914.

[6] D. Darvas, E. Blanco, and I. Majzik, "What is special about PLC software model checking? – Extended version," CERN, Report EDMS 1851093, 2017, in press. [Online]. Available: http://edms.cern.ch/document/1851093

[7] *IEC 61131-3 Programmable controllers – Part 3: Programming languages*, IEC Std., 2003.

[8] M. de Sousa, "Proposed corrections to the IEC 61131-3 standard," *Computer Standards & Interfaces*, vol. 32, no. 5-6, pp. 312–320, 2010.

[9] M. de Sousa, "Ambiguities in IEC 61131-3 ST and IL expression semantics," in *13th IEEE International Conference on Industrial Informatics*. IEEE, 2015, pp. 1312–1317.

[10] Siemens, *S7-SCL V5.3 for S7-300/S7-400*, 2005. [Online]. Available: http://support.industry.siemens.com/cs/document/5581793

[11] Siemens, *Statement List (STL) for S7-300/S7-400 Programming*, 2006. [Online]. Available: http://support.industry.siemens.com/cs/document/18653496

[12] B. Bradu, P. Gayet, and S. Niculescu, "Modeling, simulation and control of large scale cryogenic systems," in *Proc. 17th IFAC World Congress*, 2008, pp. 13 265–13 270.

[13] S. C. Park, C. M. Park, G. N. Wang, J. Kwak, and S. Yeo, "PLCStudio: Simulation based PLC code verification," in *Proc. 2008 Winter Simulation Conf.*, 2008, pp. 222–228.

[14] J. Nellen, E. Ábrahám, and B. Wolters, "A CEGAR tool for the reachability analysis of PLC-controlled plants using hybrid automata," in *Formalisms for Reuse and Systems Integration*, ser. ASIC. Springer, 2015, vol. 346, pp. 55–78.