Automatic Translation of FBD-PLC-programs to NuSMV for Model Checking Safety-Critical Control Systems

Jingyue Li* Altin Qeriqi† Martin Steffen† Ingrid Chieh Yu†

Abstract

Programmable logic controllers (PLCs) are digital control systems, commonly used in industrial automation and safety-critical applications. Control systems used in safety-critical areas must undergo an extensive and thorough certification and verification process. In safety-critical applications, the PLC programming standard IEC 61131-3 is widely accepted in industry. PLC programmers who develop control systems for safety-critical systems are often required to verify the logic of PLCs by using formal methods such as *model checking*. Translating manually from a PLC program to the input language of a model checker takes times and is often error-prone.

We develop a compiler to automatically translate PLC programs in the *function block diagram* (FBD) language, one of five industry standard PLC programming notations, to the input language of the model checker NuSMV. We have evaluated correctness, robustness, and performance of the PLC-NuSMV compiler using a case study. Evaluation results show that the compiler can translate the PLC programs correctly. The compiler can also identify several input errors and can scale to relative large PLC programs.

Keywords: model-checking, system verification, function-block diagrams

1 Introduction

Programmable logic controllers (PLCs) are computer control systems used in industrial automation. Such systems are complex and are often safety critical, meaning that failures or malfunctioning may result in loss of human life, severe environmental damage, or at least financial losses. Assuring their operational safety and mitigating critical risks is a multi-disciplinary challenge, applying to all phases of the system life cycle, and is generally heavily regulated by national or international bodies. The development of safety-critical systems rely on high and long-standing safety and engineering standards, and well-established empirical expert knowledge. To program PLCs in safety-critical systems, IEC-61131-3 [15] is one of the de-facto industry standard, that specifies five different programming languages or notations.

This project was done as collaboration with an international certification society DNV-GL (*Det Norske Veritas — Germanischer Lloyd*) and is the result of a Master thesis [21]. DNV-GL is often involved in certifying PLC-based safety-critical control systems.

*DNV-GL and Norwegian University of Science and Technology

†University of Oslo, Institute for Informatics

This paper was presented at the NIK-2016 conference; see http://www.nik.no/.

To certify safety-critical system, one formal method DNV-GL wants to pilot is *model-checking* [11][3]. Model checking explores exhaustively a model of the system under investigation in an attempt to verify it formally as correct with respect to a specification. The specification is often presented using a truth table. One BDD-based symbolic model checking tool is a model checker called NuSMV [8] [10].

The use of model-checking to analyze PLC-based control systems, however, is hampered by the fact that, for a given PLC-based design, the *formal modeling* is actually often done manually. This not only requires a non-negligible effort, but also a non-trivial amount of specialist knowledge and familiarity with the model checker tool. With the high demands on robustness of the PLC software, a case-by-case manual treatment is not cost-effective. So, to integrate model-checking better into the development and certification process, this work presents a *compiler* from one of the five official PLC languages, namely FBD ("function block diagrams"), to the input language of the NuSMV model checker. Besides that, we apply the compiler to a case study "Falcon controller", which specification is publicly available [16]. The compiler does currently not cover the whole FBD language, but to the extent needed for the case study. The compiler is designed to run on standard platforms, using the cross-platform application framework Qt.

The rest of the paper is organized as follows. We introduce necessary background of PLCs and model checking in Section 2, we sketch the design, structure, and implementation of the PLC-NuSMV compiler in Section 3. Section 4 evaluates the robustness and performance of the PLC-NuSMV compiler on a case study. We conclude in Section 5 by discussing related and future work. More details of the tool and the case study can be found in the Master thesis [21].

2 Background

This section gives a brief overview of the concepts and terms relevant to this work, discussing PLCs, model checking, and also the case study.

2.1 Related work

Formal methods, as one important aspect of assuring correct system functioning, gained traction in industry. Studies [14, 5] present extensive overviews over industrial use of formal methods. As PLCs are often used in safety-critical sectors of industry, it should be not surprising that various formal methods have been applied and tailored towards the verification of PLCs and PLC programs. A study [13] gives a short overview over formal and semi-formal methods specifically for the validation and verification PLCs. Similarly study [17], concentrates on SFC and LD (sequential function charts and ladder diagrams). In that area, the considered (sub-)systems are often hardware-close, of relatively simple and rigid structure, and relatively homogeneous (at least compared to some of nowadays software applications running on distributed, heterogeneous, dynamic platforms, building upon layers of abstractions and composing various artifacts, written by many people and in different languages, and communicating and interacting in various ways). That makes PLC programs well-suited for automated, formal verification techniques such as *model checking*.

A study [23] uses equivalence checking based on boolean satisfiability (SAT) for the verification of PLC programs written in the instruction lists IL language. Study [7] presents an approach of verifying so-called *simple* programs written in (a significant fragment of) the IL language via model-checking, using Cadence SMV. "Simple" refers

to restrictions on available data types, —booleans and bounded integers, only— and considering only single-module programs. The restrictions on the data types are similar to the ones in the case study in this paper.

A study [1] presents a compiler from ST (structured text) to NuSMV's modeling language.The framework is extended in [2, 12] covering other PLC languages, as well. An overview of model checking PLCs is presented in [18].There are also compilers (e.g.[19]) that can automatically convert PLC code into input of NuSMV's. We decided to develop a compiler ourselves from scratch because a certification body (e.g. DNV-GL) needs a tool that is simple but very reliable when using the tool to certify safety-critical systems. If the tool is not reliable, results of certification can be misleading. If the tool is too complex, it will be difficult to verify reliability of the tool itself.

2.2 Programmable Logic Controllers

A *programmable logic controller* (PLC) is a specialized digital industrial control system used in automation, i.e., for operating equipment such as machinery, processes in factories, communication switches, and several other types of control systems.

PLCs were originally developed in the late 60s to replace electromechanical relay-based control systems [9]. The old and complicated systems were inflexible and they often had to be rewired or completely replaced every time the production requirements changed and the systems had to be adapted or updated. Unlike these systems, PLCs could be programmed easily with dedicated programming languages. This flexibility was a big motivation to replace the old machines with microprocessor based programmable logic controllers.

PLCs consist usually of a single microprocessor (CPU), memory, and electrical input/output-ports [6]. The ports are connected to sensors (receiving environmental input) and actuators (effecting the environment using valves, motors, relays etc). Compiled PLC programs are typically kept in the non-volatile memory of the programmable controller, and consist of series of instructions which are executed sequentially on each CPU.

PLC scan cycle

As typical for reactive systems, PLC programs are usually executed cyclically. Each *scan cycle* consists of 3 steps [16]: 1) Reading the input and storing it in a specific area of memory reserved for inputs. 2) Executing PLC code as reaction to the input, thus generating a new internal state and new output values [15], again using a dedicated area of memory, before 3) passing the output values to the actuators. The time to complete an entire cycle is known as the *scan time*, typically a few milliseconds. After the output values are passed to the actuators, a new cycle starts.

PLC program structure

A PLC program consists of encapsulated blocks of code called *POUs* (program organization units) defined by the IEC 61131-3 standard. A POU can be compiled independently and can be linked together with other POUs to form a complete program. This independence of POUs makes it possible to reuse them in different PLC programs and projects. There are three types of POUs: *functions*, *function blocks*, and *programs*. All three types consist of a *declaration* part where the input and local variables are declared and a *code* or *instruction* part where we find the program instructions. They are similar to subroutines in general purpose high-level programming languages in that they can be called with arguments.

Functions are the simplest type of POUs. They take input parameters and return an output value. Functions are "state-less" in that they can not retain their data, i.e., the values declared in this POU are lost when the function has finished executing its code. Functions can not access external/global variables, that is, variables declared outside its POU. This means, that functions invoked with the same input parameters yield the same output value.

Function blocks can be thought of as both a function and as an object when comparing them to object-oriented general purpose programming languages. They can have static variables which will not lose their values when the function blocks have finished executing. Function blocks can access external/global variables.

Programs represent the main POUs in PLC programs. They are very similar to function blocks. The only difference between function blocks and programs is that in programs global and external variables can be declared and variables declared in programs can be assigned to physical addresses like memory addresses for PLC inputs and outputs.

IEC 61131-3 and PLC languages

The International Electrotechnical Commission (IEC), an international standards organization for all fields of electrotechnology, published the common standard IEC 61131-3 for PLC languages [15]. The standard serves as a guideline for PLC programming but PLC manufacturers are not expected to implement the entire standard. There can still be some differences between programming systems but projects should easily be ported from a programming system to another. The IEC 61131-3 standard states that functions and functions blocks have to be hardware-independent as far as possible to achieve reusability across PLC projects and vendors.

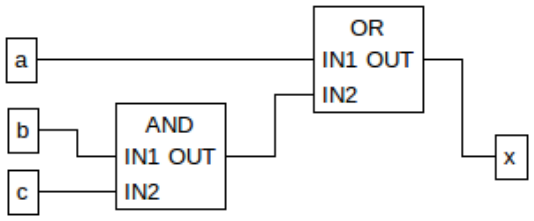
The IEC 61131-3 standard provides five different programming languages, namely *instruction list* (IL), *structured text* (ST), *ladder diagram* (LD), *function block diagram* (FBD) and *sequential function chart* (SFC) [15]. IL and ST are textual programming languages and the rest are graphical programming languages. Programs in graphical languages are represented by diagrams, representing the connections between POUs with other POUs or external variables. Textual languages are like common (rather low-level) programming languages where declarations and instructions are typed in textual form. Some of these languages are more suited for specific kind of control tasks and application areas.

Instruction list (IL) is a low-level machine-oriented language similar to assembly languages. Figure 1(a) shows a very simple example of a boolean function written in IL. IEC 61131-3 defines many basic standard functions (e.g., `SQRT`, `LOG`, `ADD`, and `COS`) and a few standard function blocks (e.g., counters and timers).

A *function block diagram* (FBD) is a type of graph where the nodes represent variables, functions, or function blocks, and the edges represent the connections between the variables in the graph and the input/output variables in POUs. FBDs are sometimes divided into several *networks* which makes it easier to structure the control flow of POUs. Like, IL, PLC programs in FBD contains a declaration part and a code part. The declaration part is usually written in textual form and the code part is the actual graph. Figure 1(b) shows the same boolean function in the function block diagram language. The two big boxes represent functions, indicating also their parameters and the output variable(s).

```
FUNCTION FuncTest: BOOL
VAR_INPUT
   a: BOOL
   b: BOOL
   c: BOOL
END_VAR
   LD a
   OR( b
      AND c
   )
   ST FuncTest
END_FUNCTION
```

(a) Represented in IL



(b) Represented in FBD

Figure 1: Boolean expression $x = a \vee (b \wedge c)$

2.3 Model checking

Model checking [11][3] is a formal method which is especially attractive for verifying the correctness of hardware circuits and hardware-close system descriptions. Model checking, generally, amounts to exhaustively and automatically check if a model of a system satisfies a given specification. The model describes the system behavior in a mathematically precise and unambiguous manner and the verification is done by systematically checking all reachable states of the model. Many different techniques and tools have been developed and used for different application areas and languages. sometimes in combination with other formal analysis techniques.

Different techniques have been investigated and implemented to deal with the notorious state-space explosion problem. One successful approach is known as *symbolic* model-checking. In contrast to explicit-state model checkers, sets of system states and transitions between those sets, are presented symbolically, in particular using efficient (symbolic) graph-representation of binary functions, *binary decision diagrams* (BDDs). We make use of NuSMV [8] [10], a prominent state-of the art representative of BDD-based symbolic model-checkers. Specifications in SMV and its variants are expressed in temporal logic, classically in CTL (computation tree logic), but NuSMV supports also linear-time temporal logic (LTL), which is what we use in our compiler. LTL in NuSMV is extended with past operators, but these operators are not relevant to our compiler.

For a flavor of the NuSMV input language, Figure 2 shows the representation of the (rather trivial) example used in Figure 1.

2.4 The case study

As mentioned, we use the case study "Falcon-controller" as described in a technical report by Matti Koskimies [16]. The case study in that report is an example of translating PLC code to the input language of NuSMV. The PLC programs are quite simple in that they can only contain Boolean variables, Boolean function calls (which are basically logic gates, e.g., *and*, *or* and *xor*), connections between the logic gates and input/output variables. The PLC program is a simplified PLC programs a certification body (e.g. DNV-GL) may see in its certification work.

Figure 3, taken from [16], shows the PLC logic of the case study and describes the control logic of one of the components in the Falcon protection system which is used to protect electrical instrumentation and switchgear from electric arcs. We can see in this figure several input ports (`CH1–CH4` and `F8.01–F8.16`), output ports (`Triac 1–4` and `Relay 1–6`) and the logic gates `AND` (&) and `OR` (≥ 1).

```
MODULE FBD_Program(a, b, c)
VAR
    x : boolean;
DEFINE
    and_gate0    := b & c;
    or_gate0     := a | and_gate0;
ASSIGN
    init(x)      := FALSE;
    next(x)      := or_gate0;
-----------------------------------
MODULE TruthTable(a, b, c)
VAR
    x : boolean;
ASSIGN
    init(x)      := FALSE;

    next(x)      :=
        case
            !a & !b & !c     : FALSE;
            !a & !b &  c     : FALSE;
            !a &  b & !c     : FALSE;
            TRUE             : TRUE;
        esac;
```

(a) Module declarations

```
MODULE main
VAR
    a : boolean;
    b : boolean;
    c : boolean;

    fbd : FBD_Program(a, b, c);
    truth_table : TruthTable(a, b, c);
ASSIGN
    init(a) := {FALSE, TRUE};
    init(b) := {FALSE, TRUE};
    init(c) := {FALSE, TRUE};

    next(a) := {FALSE, TRUE};
    next(b) := {FALSE, TRUE};
    next(c) := {FALSE, TRUE};

-- Specification --
LTLSPEC G (fbd.x <-> truth_table.x)
```

(b) Main module and spec.

Figure 2: An example of a NuSMV program

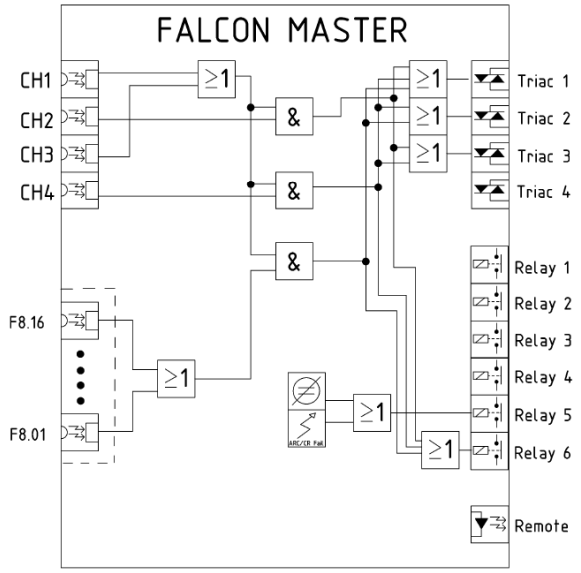


Figure 3: The PLC logic of the master unit of the Falcon system [16]

3 PLC-NuSMV compiler

This section will sketch the design and implementation of the PLC-NuSMV compiler.

Modular design: The compiler follows established practice, separating the compiler into a sequence of distinct phases with clearly define interfaces. See Figure 4 for an overview over the phases.

Extensibility: The implementation currently supports only (a subset of) the FBD language, basically to the extent as needed for the case studies. In important goal, hand in hand with the modular design and in particular distinguishing front-end and back-end, is the possibility to extend the compiler, either with further PLC languages and/or other model-checking back-ends.

Portability: The implementation is done in C^{++}, making use of the widely used cross-platform application framework. The compiler thus runs on all platforms on which the NuSMV-compiler is available (Linux/Windows/MacOS).

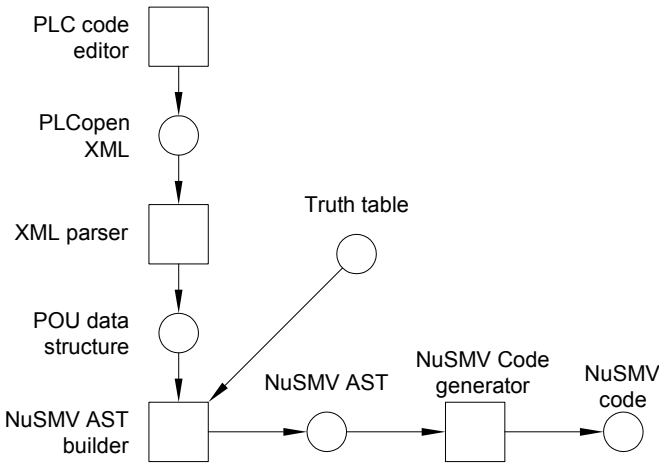


Figure 4: Schematic architecture and compilation phases of the compiler

Standard compliance and openness: In line with the other goals, the compiler takes as input PLC-descriptions in the open exchange format PLCopen [20] (a "domain-specific" XML format), which is supported by many PLC manufacturers. In this work, we use the PLC graphical editor Beremiz [4] for describing PLC programs, which generates PLCopen-code, but the compiler is independent from the choice of the editor, as long as it supports the open XML-standard for PLCs.

4 Evaluation

In this chapter, we will discuss the efficiency and the correctness of the compiler. We will also discuss some scalability limitations of the compiler. The scalability issue is common in general in model-checking approaches.

4.1 Validation of the compiler

The case study from the technical report by Matti Koskimies [16] is used as a benchmark to test the practicality and usability of the compiler. It was used also to validate that the compiler produces the correct NuSMV code. In the case study, the PLC program from Figure 3 has been translated to a NuSMV model and then verified by the NuSMV tool. The PLC-NuSMV compiler should translate from a PLC program in the FBD language that is equivalent to the PLC logic of the case study, to a NuSMV model that is the same as the one in the same case study. Before being able to verify the compiler with the PLC program from the case study, the PLC program was first translated to the FBD language by hand.

The case study was not the only PLC program example that was created to verify the correctness of the compiler. Several other PLC programs were also created and their correct translation manually verified against "expected" SVM models. Several simple PLC programs containing various combinations of functions were created to verify that they were correctly implemented.

Results

All the PLC programs created by hand were translated by the PLC-NuSMV compiler to the expected NuSMV models. Every NuSMV model description produced by the

compiler was carefully and thoroughly reviewed before the final verification with the NuSMV model checker. NuSMV confirmed that the LTL specification in all the NuSMV models were true in all states.

The testing of the case study and all other PLC program examples confirmed that the compiler can correctly translate PLC programs in the FBD language to the input language of NuSMV.

4.2 Robustness

The ability for a program to cope with errors and erroneous input is critical for a robust program. The PLC-NuSMV compiler should display intuitive and clear error and warning messages to help the user localizing and ultimately repairing the error.

A class was implemented to simplify error messaging in the program. It is used mainly for displaying error and warning messages to the user of the program. This centralization of logging and message displaying makes it easier to extend the software. For example, if the program is integrated with a PLC code editor, the error message can be sent to the editor which can then show them. For an overview of possible extensions and improvements to the compiler, see Section 5.

Command-line option validation

The compiler validates the arguments to the command-line options it receives on start-up. If it detects invalid input, it gives the user an error message. It will also suggest to the user to use the `--help` command-line option. This option contains usage information about all the possible command-line options for the PLC-NuSMV compiler. Both the PLC program and the truth table files must be given as arguments to the program. The program informs the user about that if the PLC program or the truth table is missing.

Detection of errors during execution

The PLC-NuSMV compiler validates all the files given as input to the program. It tries first to read the file that contains the PLC program and the truth table file. If they are not readable, the compilation process will abort and an appropriate error message will be displayed. The XML file is then validated against the PLCopen XML schema. The XML schema file should be included with the compiler. If the XML file is valid, the truth table file is then validated. The compilation process aborts if any of these files are not valid. During the parsing of the XML file, the program can detect logical errors, e.g., the PLC program does not contain output variables or it does not contain one *program* element. It's not possible to proceed the compilation process for most of theses cases, so the compiler displays an error message and aborts.

The compiler ignores some non-critical errors, e.g., the input variables of a POU function is not connected to anything or an output variable is not connected to a POU function block or an input variable. In these cases, the program simply just gives a warning message to the user.

Most of the possible errors should be detected during the parsing of the input files, but there are some cases where this doesn't happen before the actual translation of PLC code to NuSMV starts. Most of them are related to cases where the truth table does not correlate with the PLC program. Before the NuSMV code generations starts, all errors related to the PLC program and the truth table has been found. Only the inability to write the NuSMV code to a file can cause the program to abort.

There are some errors the user may not be able to do anything about without changing the source code. For example, a PLC program might contain function blocks or unsupported Boolean functions. An appropriate error message is displayed for some of these situations.

4.3 Performance evaluation

We evaluated not only the performance of the compiler itself, but also the resulting efficiency of the approach as a whole, i.e. evaluating the efficiency of the used NuSMV-model checker on the used test inputs. Besides that, we implemented a small "test-case generator", which allowed to generate *"synthetic"* test cases, i.e., schematic FBD programs whose size can be scaled arbitrarily. The critical parameter for the size of those programs is the number of input variables, as it determines the size of the state space. For profiling we used the *Valgrind Function Profiler*, or *callgrind*. It was was used to get a detailed statistics, e.g., over the number of instructions executed on each function and how many times each function where called.

Test bed

The experiments have been done in an Intel Core i5-3570 3.4 GHz processor and 8 GB of RAM, under Linux, Debian 8. The timing of all test runs were done with the command-line tool *time*.

Results end evaluation

All the PLC programs created manually including the case study, are compiled very quickly. It takes about 100 milliseconds for the PLC-NuSMV compiler to read the case study PLC program example, translate it to NuSMV models and write the NuSMV code to a file.

When running the compiler with the test case generator option, the compilation time increases linearly with the size of the input program. That is to be expected, as the compiler currently does not perform any extensive optimizations. As mentioned, the crucial parameter for the synthetic test cases is the number of variables. The test cases mimics the spirit of the case study insofar that the *specification* of the PLC program is given in the form of a truth table. That, however, means that not just the size of the logical state-space increases exponentially —an instance of the well-known state-space explosion problem— but also the model *description* in the form of a exponentially growing truth-table. This the compilation time grows exponentially with the number of input variables (i.e., linear in the size of the program description). The same holds the amount of memory used by the compiler. Most of the processing time (about 87% for 17 input variables and 1 output variable) seem to be the part of the program where the truth table is created by traversing the graph of the POU data structure.

The time to generate the NuSMV model seem to also increase exponentially. About 10% of the total processing time is related to writing the NuSMV code. The reason for this is because each of the switch-case expressions in the truth table module of the NuSMV model contains part of the truth table.

Possible performance improvements

Instead of creating a truth table to use as a specification for a PLC program, we can use Boolean formulas/functions. The compiler has been implemented with the ability of

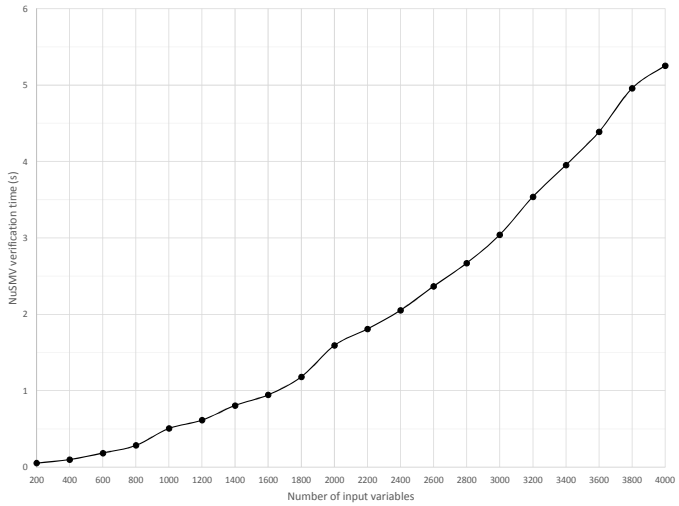


Figure 5: NuSMV verification times

skipping the creation of the NuSMV truth table module and instead use Boolean formulas to create the specifications to verify PLC programs. The option −o (*--optimize*) makes the compiler do exactly this.

As expected, the memory consumption and the execution time of the PLC-NuSMV compiler now increases linearly in the number of input variables.

The compiler and the NuSMV model checker has only been tested with up to 10 thousand input variables. It takes about 1 minute to compile a PLC program with that amount of input variables.

Running NuSMV

The time and memory consumption for *verifying* such NuSMV models with the model checker also doesn't increase as it did with the truth table approach of verification. The chart in Figure 5 shows the execution time it took NuSMV to verify various synthetic PLC programs generated with the built-in optimization option. 20 tests were done. In each new test, the input variable was increased by 200. Based on the tests we ran, it looks like the verification time increases linearly as the number of input variables increases. It increases faster than the compilation time of the PLC-NuSMV compiler.

This modification to the compiler was done to show that using the truth table approach fails after a certain level of complexity of PLC programs. To verify very large real-life PLC program examples, the compiler can be extended with the ability to takes Boolean formulas as input.

5 Conclusion

The paper reported the design, implementation, and evaluation of a compiler from the FBD-PLC programming language to the input language of NuSMV. The compiler has been thoroughly tested with (automatically generated) PLC programs to test its robustness and performance, The compiler, while handling the case study and the range of test cases adequately is still a proof-of-concept: it currently supports only the part of FDB. For PLC programs that are similarly sized to the one in the case study, or slightly bigger, the PLC-NuSMV compiler itself shows adequate run-times, and also the model-checker is able to analyze the resulting model. The size of PLC programs is limited by the choice of expressing the *specifications* of PLC programs in the truth-table approach.

As for future work, we plan to extend the range of features supported by the compiler. Similarly, one increases the applicability by re-targeting the front-end to support other PLC languages. We suspect, though, that NuSMV will still be a natural choice for PLC programs in the targeted application domain, since BDD-based model checking has shown its strength in particular in hardware-close applications. Besides increasing the scope of the tool, the compiler itself can be improved. As of now, the compilation is rather straightforward. While taking care that the user is warned about questionable inputs (such as unused variables and the like) and adequately informed about errors about (yet) unsupported features. There should be room for optimizing the compilation. It is known that the efficiency of BDD-based model checking is sensitive to the actually (internal) representation of the model, in particular the variable order used for the BDD-representation. Different logically equivalent representations can differ, in the worst case, in their performance. Further experiments would be needed to find heuristics or compilation strategies, which, in conjunction with NuSMV, leads to efficient model checking, at least on the average case. Methodologically, the case study shown in [16] uses a *truth-table* approach: the desired behavior of the control program is given via tabulating the boolean input-output behavior. While that might seem natural enough, it is certainly not a scalable approach (independent from the aspect of model checking). It also raises the question in which way the specification itself, being a big truth table, is free of errors. It also seems that spelling out the specification in such a combinatorial manner might prevent NuSMV to play out its strength of compact and efficient symbolic representations of models, but further experiments would be needed here. In any case, we consider a more symbolic way of specifying the behavior of the program as advantageous. Another our future work is to compare our compilers with similar ones (e.g. [19]) quantitatively and qualitatively, in order to improve our compiler further.

References

[1] B. F. Adiego, D. Darvas, J.-C. Tournier, E. B. Viñuela, , J. O. Blech, and V. M. G. Suárez. Automated generation of formal models from ST programs for verification purposes. Internal Note CERN-ACC-NOTE-2014-0037, CERN, 2014.

[2] B. F. Adiego, D. Darvas, E. B. Viñuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez. Applying model checking to industrial-sized PLC programs. *IEEE Transactions in Industrial Informatics*, 11(6), Dec. 2015.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, May 2008.

[4] Beremiz home page, 2016. `http://www.beremiz.org`.

[5] J.-L. Boulanger, editor. *Formal Methods: Industrial use from Model to the Code*. Wiley, 2012.

[6] L. W. Brittian. Programmable logic controllers. In *Audel Electrical Trades Pocket Manual*, chapter 10, pages 89–97. John Wiley & Sons, Inc., Hoboken, New Jersey, April 2012.

[7] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen. Towards the automatic verification of PLC programs written in instruction list. In SMC [22], pages 2449–2454.

[8] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. *NuSMV 2.5 User Manual*, 2011.

[9] P. Chevtsov, S. Higgins, and D. Seidman. *PLC Support Software at Jefferson Lab*, October 2002.

[10] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV version 2: BDD-based + SAT-based symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Proceedings of CAV '02*, volume 2404 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[11] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[12] D. Darvas, I. Majzik, and E. B. Viñuela. Formal verification of safety PLC based control software. In E. Ábrahám and M. Huisman, editors, *Proc. of the 12th International Conference on integrated Formal Methods (iFM 2010)*, volume 6396 of *Lecture Notes in Computer Science*, pages 508–522. Springer Verlag, 2016.

[13] G. Frey and L. Litz. Formal methods in PLC programming. In SMC [22], pages 3333–3339.

[14] M. G. Hinchey and J. P. Bowen, editors. *Industrial-Strength Formal Methods*. International Series in Formal Methods. Springer Verlag, 1999.

[15] K.-H. John and M. Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems*. Springer Verlag, 2010.

[16] M. Koskimies. Applying model checking to analysing safety instrumented systems. Research Report TKK-ICS-R5, Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland, June 2008.

[17] S. Lamérière, O. Rossi, J.-M. Roussel, and J.-J. Lesage. Formal validation of PLC programs: A survey. In *1999 European Control Conference (ECC)*, 1999.

[18] T. Ovatman, A. Aral, D. Polat, and A. O. Ünver. An overview of model checking practices on verification of PLC software. *Software System Modelling*, 2014.

[19] O. Pavlović and H.-D. Ehrich. Model checking PLC software written in function block diagram. *International Conference on Software Testing, Verification and Validation*, 00:439–448, 2010.

[20] PLCopen XML, 2016.

[21] A. Qeriqi. A PLC-NuSMV compiler for model checking safety critical control systems. Master's thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, May 2016.

[22] *Proceedings of the IEEE Intl. Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2000.

[23] A. Sülflow and R. Drechsler. Verification of PLC programs using formal proof techniques. In *FORMS/FORMAT 2008*. L'Harmattan Hongrie, 2008.