Direct Model Checking of PLC Programs in IL

Bastian Schlich * **Jörg Brauer** * **Jörg Wernerus** *
Stefan Kowalewski *

* *Embedded Software Laboratory, RWTH Aachen University, 52074 Aachen, Germany (e-mail: lastname @ embedded.rwth-aachen.de)*

Abstract: While there are several approaches applying model checking to PLC programs, it is still not used in industry. This is due to the limited applicability of the existing approaches, which all translate PLC programs into the input languages of existing model checkers and thus suffer from certain problems. This paper presents a new approach that applies model checking directly to PLC programs written in IL without using translations. This has some advantages: domain-specific information is available during verification, users can make propositions about all features of the PLC, and counterexamples are given in the same language as the program, thus, simplifying the process of locating errors. In the described approach, a tailored simulator builds the state space for verification. Within this simulator, different abstraction techniques are used to tackle the state-explosion problem. A case study shows the applicability of this approach.

Keywords: Programmable Logic Controllers, Instruction List Programs, Formal Verification, Model Checking.

1. INTRODUCTION

There are several approaches that apply model checking (Clarke et al. (1999)) to Programmable Logic Controller (PLC) programs. They all work in a similar manner by translating PLC programs into the input language of an existing model checker. Nevertheless, model checking is still not used for the verification of PLC programs in industry due to the limited applicability of the existing approaches.

This paper presents a new approach, which applies model checking directly to PLC programs written in Instruction List (IL) without using translations. This has some advantages. The model checker can use domain-specific information such as the cycle of the PLC during model checking to reduce state spaces. Furthermore, users can make propositions about all features of PLCs within the formal specification. Another advantage is that counterexamples are shown directly in IL. Thus, users can more easily locate the source of an error, and the counterexample gives precise information about how the property violation emerged.

The fundamental concept in our approach is to build state spaces for model checking using tailored simulators that directly work on IL programs. These simulators use domain-specific information and apply different abstraction techniques to tackle the state-explosion problem (Clarke et al. (2001)). We have implemented this approach in our model checker [MC]SQUARE. To make it applicable, [MC]SQUARE supports the complete instruction set defined in the IEC 61131-3 standard (International Electrotechnical Commission (1993)) and automatically applies abstraction techniques without any manual intervention. Moreover, users do not have to prepare the IL programs to be checked. Additionally, [MC]SQUARE features a GUI, which is similar to the GUI of a debugger.

After a presentation of related work in Sect. 2, a general introduction of our approach is given in Sect. 3. In this approach, state spaces for model checking are built using tailored simulators. Two of these simulators are presented in Sect. 4. Sect. 5 shows the applicability of our approach using four different case studies.

2. RELATED WORK

There are many approaches that deal with the application of formal verification techniques to PLC programs. Verification techniques used in these approaches include static analysis, theorem proving, and model checking. As this paper describes an approach that applies model checking, this section presents related work regarding model checking. All approaches known to us use a translation approach, which translates the PLC program into the input language of an existing model checker. The approaches presented in this section all use SMV, CADENCE SMV, or NUSMV for model checking.

Moon (1994) describes an approach to verify PLC programs given in the language Ladder Diagram (LD). In this approach, the programs given in LD are translated into the input language of SMV. In the translation, the rungs of the alternative paths are translated into boolean assignments. This approach supports only a limited amount of constructs, namely switches and coils and their negated versions. Another problem is that during the translation no abstraction techniques are used, which leads to the state-explosion problem.

Canet et al. (2000) present an approach that verifies PLC programs given in the language IL. The IL programs are translated into the input language of SMV. In this translation process not all IL instructions are supported. The PLC cycle is not explicitly modeled in this approach. This led to a problem in the case study described by Canet et al. (2000), who applied model checking to a program controlling a turning center. In this case study, a formula was refuted because an error was found in the middle of the execution of the PLC cycle. This was a false alarm because in the middle of the execution of the PCL cycle, the behavior is not visible to the outside. To fix this problem, Canet et al. (2000) had to include the program counter in the specification, that is, they specified that the specification has to be valid at certain program locations only.

Mertke and Frey (2001) present an approach that combines modeling and verification. In this approach IL programs are translated into Petri nets. As in the other approaches, not all features of IL programs are handled. This Petri net together with two other Petri nets, which are manually created and represent the PLC and its environment, are translated into the input language of SMV. The specification can be expressed using CTL or a language called Sicherheitsfachsprache (Mertke (2004)).

Huuck (2003) describes an approach that translates PLC programs written as a Sequential Function Chart (SFC) into the input language of CADENCE SMV. The approach does not handle all constructs found in SFCs because some of these constructs have an ambiguous semantics. Therefore, Bauer and Huuck (2002) defined a subset of SFCs as safe SFCs, which have a well-defined semantics. This set of constructs is supported by this approach.

Pavlovic et al. (2007) developed an approach that translates PLC programs given in the language Statement List (SL), which is a PLC language developed by Siemens, into the input language of NUSMV. SL is similar to IL, but the syntax is a little different and it has some additional instructions to access certain features of the Siemens PLCs directly. The hardware model that is used in this approach is that of a Siemens S7. Without manual adaptation of the generated NUSMV model, verifying specifications is often too time-consuming (cp. Sect. 5.1).

3. APPROACH

[MC]SQUARE (Schlich (2008)) is a model checker, which was developed to model check microcontroller assembly code. It is written in Java and features a graphical user interface. [MC]SQUARE supports the ATMEL ATmega16, the ATMEL ATmega128, the Intel MCS-51, and the Infineon XC167 microcontrollers. Several abstraction techniques are available in [MC]SQUARE such as path reduction and dead variable reduction (Schlich et al. (2008b)) and delayed nondeterminism Noll and Schlich (2008).

This paper describes the extension of [MC]SQUARE to model check IL programs for PLCs. To be able to apply [MC]SQUARE to real-world applications, it supports the complete set of instructions of the IEC 61131-3 standard including all functions and function blocks from the standard library in the programming system CODESYS.

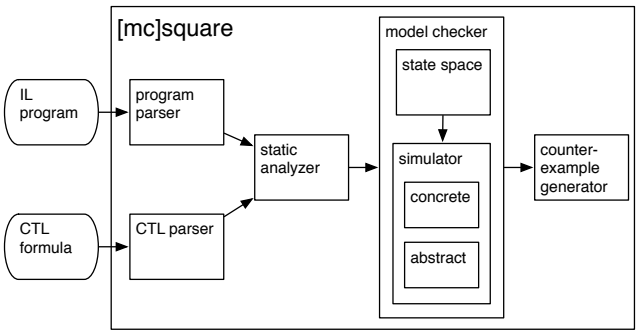

Fig. 1. Model checking process applied in [MC]SQUARE

Additionally, self-defined functions and function blocks are supported.

The model checking process that is applied within [MC]-SQUARE is shown in Fig. 1. Our tool requires as input an IL program and a formula given in Computation Tree Logic (CTL) (Ben-Ari et al. (1983)). [MC]SQUARE reads IL programs written for CODESYS, with one source file for each function. It is sufficient to indicate the main program and [MC]SQUARE automatically loads all dependent functions and function blocks.

The program parser and the CTL parser transform the program and the formula into their internal representations. Next, static analyses are conducted, which deliver information for the simulator by examining the program code. In order to verify a given specification, the model checker evaluates the state space of the analyzed program, which is built using a tailored simulator (see Sect. 4). New states are directly stored in the state space and accessed by the model checker.

In [MC]SQUARE, we have implemented the local model checking algorithm of Heljanko (1997), which is based on the approach of Vergauwen and Lewi (1993). Local algorithms determine the truth values of the formula for the initial state only, and thus, can be applied on-the-fly. In contrast, global algorithms determine the truth values of the formula for all states.

For verification of PLC programs, the user chooses either the concrete or the abstract simulator for state space building. Independently of the simulator used, [MC]SQUARE generates a counterexample if the specification is violated. Multiple ways exist to visualize counterexamples in [MC]-SQUARE, supporting the developer in understanding the defect found.

Additionally, [MC]SQUARE has a feature to simulate program execution. The simulation process can be controlled by the user. Input assignments can be determined before execution of the next step, which helps the user in reproducing incorrect program behavior. Moreover, it is possible to simulate multiple steps both forward and backward, which distinguishes [MC]SQUARE from conventional debuggers and renders it a helpful tool for debugging, as well.

4. DESCRIPTION OF THE SIMULATORS

Model checking is not performed on the system itself but on the state space of the system. In [MC]SQUARE, the state space is built by a tailored simulator. Developing

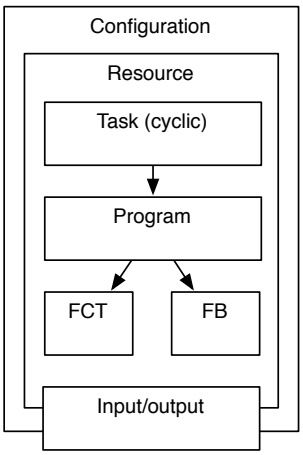

Fig. 2. Model of a conventional PLC (Lewis (1998))

simulators is the most important part in our research. The simulators execute the IL program based on the underlying hardware model of a conventional PLC.

The model of a conventional PLC (Lewis (1998)), on which most existing PLCs are based, is displayed in Fig. 2. The conventional PLC consists of one resource, one cyclic task, and one program. The cyclic task executes in cyclic operation mode, which comprises the following steps: reading input, executing the program, and writing output. The program is allowed to use functions, function blocks, inputs, and outputs. Output values are written only at the end of a cycle.

For the verification of IL programs, we have developed two simulators: a concrete and an abstract one. While the concrete simulator operates on single values, we have developed an abstract simulator, which performs computations on intervals to tackle the state-explosion problem. Both simulators model the cyclic operation mode of the conventional PLC. [MC]SQUARE stores newly created states only after the outputs are written. This approach reduces state spaces by storing states only after a complete cycle was executed. All memory configurations that are not visible are omitted. Moreover, [MC]SQUARE only verifies the written output values.

The following operations are performed by the simulator in order to create the successors of a state:

(1) Determine all possible input assignments.
(2) For each input assignment, simulate a complete cycle of the PLC program and evaluate the truth values of the atomic propositions.
(3) Store the resulting states in the state space.

[MC]SQUARE abstracts from environment behavior. In order to create an over-approximation of all possible behaviors of the PLC program, the simulator has to execute the program with all possible input assignments. The concrete and the abstract simulator differ in how these input assignments are dealt with. The concrete simulator is executed once for each possible combination of input variables. In contrast, the abstract simulator operates on intervals of values that lead to the same control flow. Hence, a state in the abstract simulator represents a set of concrete states.

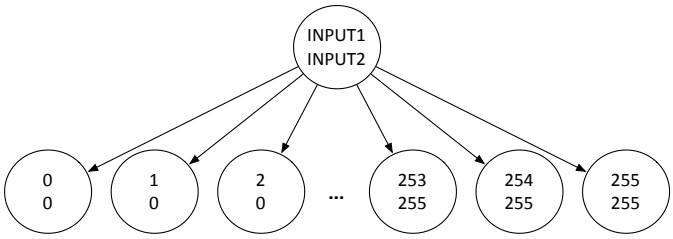

Fig. 3. Input assignments for two variables of type `BYTE`

After program simulation, the atomics of the formula are checked for the new state. Users are allowed to make assumptions about inputs, outputs, and internal variables in a formula.

Since instructions in CoDeSys do not have a formal semantics, we have validated their behavior by testing. To handle self-defined functions and function blocks, we have added auxiliary instructions `FCTCALL`, `FCTEND` and `FBEND` to mark function calls and the beginning and end of a function or function block. The main purpose of these special-purpose instructions is to initialize and to reset the accumulator before and after the simulation of a function or function block.

For the implementation, we have used the visitor design pattern to ensure that all simulators implement the complete instruction set of the IEC 61131-3 standard.

4.1 The Concrete Simulator

The concrete simulator executes the program for each possible combination of concrete input values. Before program execution, all input assignments are determined and stored explicitly. Given a program with two input variables of type `BYTE`, for instance, this leads to 2^{16} possible combinations of input values. After an input assignment is written into the memory, the simulator executes the program.

If a program uses input data types with wide value-ranges, the creation of all assignments can be too time- and memory-consuming for practical applications. Therefore, the concrete simulator sometimes suffers from the state-explosion problem. For example, for a single input of type `DWORD`, the concrete simulator has to create 2^{32} successors. Moreover, the number of states grows exponentially with the number of input variables. The situation for two input variables of type `BYTE`, for which 2^{16} possible combinations exist, is depicted in Fig. 3.

4.2 The Abstract Simulator

We analyzed which abstraction techniques that exist in [MC]SQUARE are applicable to PLCs. Techniques such as delayed nondeterminism generate successors not until the program accesses input values, for instance, in an arithmetic computation. Until the concrete values are required, the simulator executes the program with nondeterministic values, and hence, reduces the number of states by postponing the instantiation of values. Due to our modeling of the PLC cycle such techniques have no effect on the number of states because the states are only stored at the end of the cycle when the program execution is finished. This led to the development of the abstract simulator.

```
PROGRAM PLC_PRG              Slice INPUT1
VAR                          LD      INPUT1
 INPUT1 AT %IX0:BYTE;        GT      50
 INPUT2 AT %IX1:BYTE;        JMPC    label1
END_VAR
 LD      INPUT1              Slice INPUT2
 GT      50                  LD      INPUT2
 JMPC    label1              EQ      100
 LD      INPUT2              JMPC    label2
 EQ      100
 JMPC    label2
 ...
```

Fig. 4. Example program and its slices

The abstract simulator works on intervals of values instead of single values. It combines all input assignments that lead to the same program control flow to an interval and executes the program on these intervals. The abstract simulator needs a static analysis that is executed before state space building to determine which input values lead to the same control flow and thus can be combined into intervals.

During static analysis, [MC]SQUARE first constructs the control flow graph (CFG) of the program. A CFG is a directed graph in which each instruction is represented by one node. The CFG contains an edge (i_1, i_2) if and only if the execution of instruction i_1 may be followed by the execution of i_2. That means, the CFG contains all possible execution paths through the program. Only branching instructions have more than one successor in IL.

Then, [MC]SQUARE applies program slicing (Weiser (1981)) for each input variable to determine all instructions where this input variable influences the control flow. Slicing delivers a subset of the program consisting only of instructions whose execution depends on the input variable. An input variable can influence the control flow by a comparison or a type conversion before a conditional jump is executed, which requires an accumulator of type `BOOL`. The static analysis backtracks which values lead to a jump and which do not. These values are combined to intervals.

During state space building, the determinizer of the abstract simulator creates all combinations of assignments of the intervals determined during the static analysis instead of all combinations of assignments of concrete values as done by the concrete simulator. For each assignment, the program is executed and the formula atomics are checked on intervals. Similar to the concrete case, the successors are stored in the state space after the program has been executed for each assignment.

For the program shown in Fig. 4, the static analyzer divides the values for `INPUT1` into two intervals $[0, 50]$ and $[51, 255]$. If the value of `INPUT1` is in $[0, 50]$, the jump is not performed; otherwise the program jumps to `label1`. For `INPUT2` the same principle leads to the intervals $[0, 99]$ and $[101, 255]$, for which the jump is not executed, and $[100, 100]$, for which the jump to `label2` is performed.

All possible combinations of the intervals of the different input variables are then generated as shown in Fig. 5. These intervals serve as input for the abstract simulator.

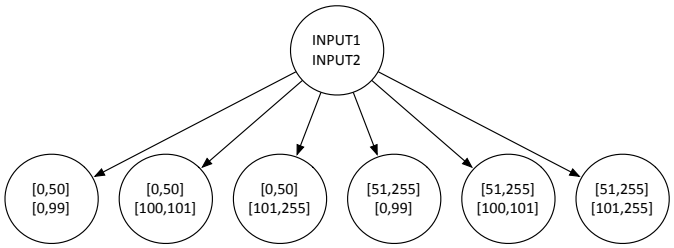

Fig. 5. Interval assignments for the example shown in Fig. 4

Overall, only 6 successors are created instead of 2^{16} for the concrete case.

5. CASE STUDIES

These case studies show the application of [MC]SQUARE to a number of different model checking problems. To evaluate the performance of [MC]SQUARE, we translated programs used to evaluate related approaches into IL and verified the same properties.

5.1 The Program DemonstrateFormByte

Pavlovic et al. (2007) demonstrated their method using the program *DemonstrateFormByte*, which converts a value given as eight input bits into a single byte. The transformation into a byte is conducted by the function *FormByte*. The following CTL specification was verified:

$$\mathbf{AG}\,(Byte = (\,b_0 + 2 \cdot b_1 + 4 \cdot b_2 + 8 \cdot b_3 + 16 \cdot b_4 + 32 \cdot b_5 + 64 \cdot b_6 + 128 \cdot b_7))$$

[MC]SQUARE required 6.3 seconds to verify this formula without any manual adaptations. It created 65,537 states, but stored only 256. All in all, 794 kB of memory were required for the state space.

In contrast, the approach of Pavlovic et al. (2007) required approximately eight hours to verify the program without manual adaptations. The verification time could be reduced to 113 seconds by manually narrowing the variable range and by preventing NUSMV from instantiating function variables outside of functions. [MC]SQUARE is capable of model checking this program efficiently without the need for manual preparations.

5.2 A Chemical Batch Plant

Huuck (2003) demonstrated his approach on a chemical batch plant shown in Fig. 6. The plant consists of two tanks A and B for raw material and a reactor C with a stirrer. Boolean inputs report to the controller when a tank is empty or full. Five valves control loading and draining of the tanks.

The plant operates as follows. First, both tanks are filled with raw material. Then V3 opens and the content of A flows completely into the reactor. As soon as A is empty, V4 opens, the content flows into the reactor, and the stirrer begins to work. After B is completely drained, the stirrer stops, the reactor is drained, and the process restarts.

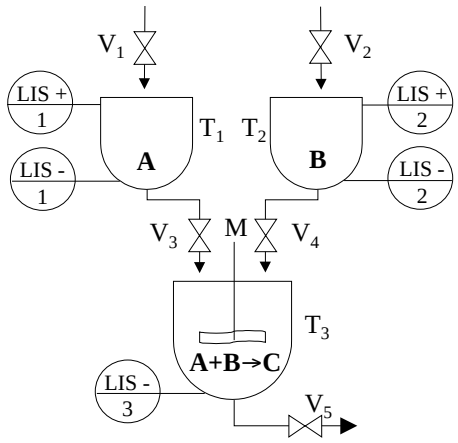

Fig. 6. Model of a chemical batch plant (Huuck (2003))

Table 1. Model checking results for the chemical batch plant

CTL formula	States stored	States created	Mem. [MB]	Time [s]
$\mathbf{AG}\neg(V1 \wedge V3)$	161	5,153	0.302	0.14
$\mathbf{EF}\, STEP = 1$	1	1	0.179	0.03
$\mathbf{EF}\, STEP = 2$	33	673	0.234	0.2
$\mathbf{EF}\, STEP = 3$	65	961	0.275	0.2
$\mathbf{EF}\, STEP = 4$	113	1,057	0.244	0.3
$\mathbf{AG}(\mathbf{AF}\, STEP = 1)$	65	737	0.252	0.2
$\mathbf{AG}(\mathbf{AF}\, STEP = 2)$	33	65	0.197	0.02
$\mathbf{AG}(\mathbf{AF}\, STEP = 3)$	33	65	0.930	0.01
$\mathbf{AG}(\mathbf{AF}\, STEP = 4)$	33	65	0.488	0.02

The results of the verification of the different formulas using [MC]SQUARE on an Intel Centrino 1.3 GHz and 512 MB RAM are shown in Tab. 1. Huuck verified all properties in a fraction of a second on a Sun ULTRA with 167 MHz.

In contrast to Huuck's approach, where the environment was modelled manually, [MC]SQUARE abstracts from environment behavior as described in Sect. 4. [MC]SQUARE generates a spurious counterexample for the last four formulas and detects that even though V1 and V2 are opened, no material flows into the tanks. Hence, they will never be filled completely. These spurious warnings, however, can be eliminated by integrating the work of Schlich et al. (2008a) into the PLC model checking in [MC]SQUARE. When we consider the difference between the used computer systems we conclude that the performance results of Huuck's approach and of [MC]SQUARE are comparable.

5.3 A Modified Chemical Batch Plant

We extended the previous program to deal with precise filling levels. The boolean inputs were replaced by inputs of type `BYTE`. These input variables indicate the current contents of the tanks in liters. We assume that tanks A and B both have a capacity of 50 liters. The plant operates the same way as before. The abstract simulator verifies the property expressed by the formula $\mathbf{AG}(\neg(V1 \wedge V3)$, which checks if there exists no state such that V1 and V3 are opened and raw material can flow from the source through A into the reactor, within 0.11 seconds. Verifying the formula required 280 kB of memory. Neither our concrete simulator nor related approaches could verify this formula. Hence, our approach is capable of model checking programs that could not be handled before.

Table 2. Results of model checking the Miller-Rabin prime test

Prime candidate	Instructions per cycle	States stored	Mem. [MB]	Time [s]
1,481	$\sim 2,362$	33,287	29.178	91.52
8,353	$\sim 3,320$	33,228	29.192	126.48
10,111	$\sim 60,744$	33,229	29.168	1873.34
14,779	$\sim 88,752$	33,330	29.174	2328.13

5.4 The Miller-Rabin prime test

As a last program, we have implemented the Miller-Rabin prime test in IL. The program uses variables of types `REAL`, `DINT`, and `STRING` as well as instructions such as type conversions, which are not supported by other approaches (cp. Sect.2).

The prime test is a probabilistic test that is based on the Fermat criterion $a^{n-1} mod\ n \neq 1$. It checks if an odd number n is prime. If and only if for every number $a \in \{2, \ldots, n-1\}$ the criterion holds then n is prime. Instead of checking each number from 2 to $n-1$ we choose a randomly and apply the test. If the test fails on a single random number, we can be sure that n is not prime. Otherwise, after i iterations we can say that n is prime with a probability of $(\frac{1}{2})^i \cdot 100\%$.

First, the IL program generates a random number $a \in \{2, \ldots, n-1\}$ using the linear congruence theorem. If the generated number is not within the bounds, the program starts a new calculation. Next, the prime test is conducted and a string variable indicates whether the test was successful or not.

For this program, we checked whether the formula $\mathbf{AG}(a > 1 \wedge a < n)$ is satisfied. The results are shown in Tab. 2. We used four numbers in this prime test: 1,481, 8,353, 10,111, and 14,779. Furthermore, we measured how many instructions were executed per cycle. In this program, each cycle corresponds to a stored state. We also measured memory and time needed for model checking. The instructions per cycle differ between cycles because random numbers that are not within the range are discarded. As in all four runs the same method for generating random numbers was used, all stabilize after nearly the same number of cycles . The small difference between the four runs are again caused by discarding random numbers that are not within the range.

This case study shows that the verification time scales roughly linear with the number of executed instructions (instructions per cycle $*$ cycles). There is, however, still place for improvement in model checking PLC programs using [MC]SQUARE. The program part containing the actual prime test is not relevant for the formula, but the analysis is computationally expensive. This shows that a more sophisticated program slicing method could be used to separate the relevant part of the program for the formula from the rest and only simulate the relevant part. Moers (2008) has implemented program slicing for microcontrollers to simulate only the relevant program parts. This methods could also be applied to PLCs.

6. CONCLUSION & FUTURE WORK

This paper describes an approach to verify PLC programs given in IL directly using model checking. In contrast to previous work on the verification of PLC programs, the complete instruction set of the IEC 61131-3 standard is supported. The state space is built using tailored simulators, which directly work on IL programs and automatically apply abstraction techniques to tackle the state-explosion problem. This has several advantages: the tailored simulators can apply abstractions that use domain-specific information, users can make propositions about all features of the PLCs, and the counterexample is shown in the IL program, which helps users to understand the counterexample and to locate the source of the indicated error.

The case studies described in Sect. 5 show the applicability of this approach. Still, the state explosion is a problem when model checking PLC programs. The case studies could show that our approach scales at least as good as the approaches described in Sect. 2, but in most cases, our approach scales better. The time needed for model checking was in most cases satisfying, and memory requirements were not a problem.

An important goal for the future is the development of further domain-specific abstraction techniques. In our work, we found out that domain-specific abstraction techniques are not always transferable to other hardware platforms. For example, the delayed nondeterminism, which is the most important abstraction technique used when model checking microcontroller assembly code, could not directly be transfered to model checking IL programs. It had to be changed and led to the development of the abstract simulator.

REFERENCES

Bauer, N. and Huuck, R. (2002). A parameterized semantics for sequential function charts. In *Semantic Foundations Engineering Design Languages (SFEDL 2002), Grenoble, France*, 69–83.

Ben-Ari, M., Manna, Z., and Pnueli, A. (1983). The temporal logic of branching time. *Acta Informatica*, 20(3), 207–226.

Canet, G., Couffin, S., Lesage, J.J., Petit, A., and Schnoebelen, P. (2000). Towards the automatic verification of plc programs written in instruction list. In *2000 IEEE International Conference on Systems, Man, and Cybernetics, Nashville, TN, USA*, volume 4, 2449–2454. IEEE Computer Society Press.

Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2001). Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, 176–194. Springer.

Clarke, E.M., Grumberg, O., and Peled, D.A. (1999). *Model Checking*. The MIT Press.

Heljanko, K. (1997). Model checking the branching time temporal logic CTL. Research Report A45, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland.

Huuck, R. (2003). *Software Verification for Programmable Logic Controllers*. Dissertation, University of Kiel, Kiel, Germany.

International Electrotechnical Commission (1993). *IEC 61131-3 Ed. 1.0: Programmable controllers — Part 3: Programming languages*. International Electrotechnical Commission, Geneva, Switzerland.

Lewis, R.W. (1998). *Programming industrial control systems using IEC 1131-3 (Revised Edition)*. Institution of Electrical Engineers, Stevenage, UK.

Mertke, T. (2004). *Formale Spezifikation reaktiver Systeme mit einer Sicherheitsfachsprache*. Dissertation, Brandenburgisch Technische Universität Cottbus.

Mertke, T. and Frey, G. (2001). Formal verification of plc-programs generated from signal interpreted petri nets. In *2001 IEEE International Conference on Systems, Man, and Cybernetics, Tuscon, AZ, USA*, volume 4, 2700–2705. IEEE Computer Society Press.

Moers, M. (2008). *Model Checking von Sensornetzwerk-Knoten mit Hilfe von [mc]square*. Diploma thesis, RWTH Aachen University, Aachen, Germany.

Moon, I. (1994). Modeling programmable logic controllers for logic verification. *IEEE Control Systems Magazine*, 14(2), 53–59.

Noll, T. and Schlich, B. (2008). Delayed nondeterminism in model checking embedded systems assembly code. In *Hardware and Software: Verification and Testing (HVC 2007), Haifa, Israel*, volume 4899 of *Lecture Notes in Computer Science*, 185–201. Springer.

Pavlovic, O., Pinger, R., and Kollmann, M. (2007). Automated formal verification of plc programms written in IL. In *4th International Verification Workshop (VERIFY'07), Bremen, Germany*, number 259 in CEUR Workshop Proceedings, 152–163. CEUR-WS.org.

Schlich, B. (2008). *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Aachen, Germany. URL `http://aib.informatik.rwth-aachen.de/2008/2008-14.pdf`.

Schlich, B., Gückel, D., and Kowalewski, S. (2008a). Modeling the environment of microcontrollers to tackle the state-explosion problem in model checking. In G. Tarnai and E. Schnieder (eds.), *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2008), Budapest, Hungary*, 27–34. L'Harmattan.

Schlich, B., Löll, J., and Kowalewski, S. (2008b). Application of static analyses for state space reduction to microcontroller assembly code. In *Formal Methods for Industrial Critical Systems (FMICS 2007), Berlin, Germany*, volume 4916 of *Lecture Notes in Computer Science*, 21–37. Springer.

Vergauwen, B. and Lewi, J. (1993). A linear local model checking algorithm for CTL. In *CONCUR'93, Hildesheim, Germany*, volume 715 of *Lecture Notes in Computer Science*, 447–461. Springer.

Weiser, M. (1981). Program slicing. In *Proceedings of the 5th international conference on Software engineering (ICSE 81), San Diego, USA*, 439–449. IEEE Press.