Automated Formal Verification of PLC Programs Written in IL

Olivera Pavlovic[1], Ralf Pinger[1] and Maik Kollmann[2]

[1] Siemens Transportation Systems,
Ackerstrasse 22, D-38126 Brunswick, Germany,
{Olivera.Jovanovic,Ralf.Pinger}@siemens.com
[2] Brunswick Technical University, Institute of Information Systems,
Mühlenpfordtstrasse 23, D-38106 Brunswick, Germany,
M.Kollmann@tu-bs.de

Abstract. Providing proof of correctness is of the utmost importance for safety-critical systems, many of which are based on Programmable Logic Controllers (PLCs). One widely used programming language for PLCs is Instruction List (IL). This paper presents a tool for the fully automated transformation of IL programs into models of the NuSMV (New Symbolic Model Verifier) model checker. For this, the tool needs a metadescription of the IL language. This broadens the scope of the software and allows the tool be used for programs written in many other low-level languages as well. Its application is demonstrated using a typical IL program, at the same time providing insights into the proposed automation of the process of formal verification of PLC programs. This automatic verification should provide a powerful analysis method with a wide industrial application.

Key words: automated verification, model checking, NuSMV (New Symbolic Model Verifier), Programmable Logic Controller (PLC), Instruction List (IL)

1 Introduction

Programmable Logic Controllers (PLCs) are a special type of computer used in automation systems. Generally speaking, they are based on sensors and actuators, which have the abilty to control, monitor and interact with a particular process, or collection of processes. These processes are diverse and can be found, for example, in household appliances, emergency shutdown systems for nuclear power stations, chemical process control and rail automation systems.

The programming of PLCs is achieved with the help of five languages, standardised by the International Electrotechnical Commission (IEC) in [IEC93]: (a) two textual languages: Instruction List (IL) and Structured Text (ST), and (b) three graphical languages: Function Block Diagram (FBD), Ladder Diagram (LD) and Sequential Function Chart (SFC). This paper focuses on the formal verification of programs written in IL, which is a low-level, machine-orientated language.

The simulation of IL programs and their automated transformation to VHDL (Very-High-Speed Integrated Circuit Hardware Description Language) are discussed in [Fig06]. The theoretical basics for the verification of IL programs can

be found in [CCL+00]. Further research on the topic was published in [PPKE07], which examines in more depth the handling of PLC hardware by the formal verification of IL programs. For this, a specific PLC is selected, although the same principles can be applied to any PLC. We present an enhancement of the works cited above describing how function calls can be handled by the PLC verification and presenting a tool for the fully automated transformation of IL programs into the NuSMV models. By applying the tool to a typical IL program, we demonstrate its successful application and show how the process of formal verification of PLC programs can be automated. Based on the lessons learnt from the tool, we also propose an improvement in the verificaton method.

The rest of the paper is structured as follows: Section 2 briefly reviews the method/formalism of model checking. In Section 3 the structure of an IL program is outlined and a detailed description of a behavioural model of the program also given. Section 4 presents the tool developed for the transformation of IL programs into NuSMV models. In Section 5 a case study illustrates the verification of IL programs. Finally, conclusions are drawn and plans for the future proposed.

2 Model Checking

Automated verification techniques such as model checking have become a standard for proving the correctness of state-based systems. Model checking is the process of checking whether a given model M satisfies a given logical formula φ. Model checking tools such as SPIN [Hol97] and SMV/NuSMV [McM96,CCB+02] incorporate the ability to illustrate that a model does not satisfy a checking condition using a textual, tabular or sequence chart-like representation of the relevant states.

The model M has to be translated into the input language of a model checking tool. For this, a state transition system can be used, which defines a kind of non-deterministic finite state machine representing the behaviour of a system. The transition system can be represented by a graph whose nodes represent the states of the system and whose edges represent state transitions. A state transition system is defined as follows.

Definition 1. *State transition system*
A system $T = (\mathcal{S}, \mathcal{S}_0, \rightarrow)$ *with*

- $\mathcal{S}$*: a non-empty set of states,*
- $\mathcal{S}_0 \subseteq \mathcal{S}$*: a non-empty set of initial states,*
- $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$*: a transition relation which is defined for all* $s \in \mathcal{S}$

is called a state transition system.

In the sections below we will limit ourselves to transition systems and temporal logic formulas, which are both suitable for capturing the behaviour of our programs and representing typical checking conditions. The model checking problem can be stated as follows: Let a checking condition be given by a temporal logic formula φ, and a model M with an initial state s, then it must be decided

$$M, s \models \varphi.$$

If M is finite, the model checking is reduced to a graph search. In our case, Linear Temporal Logic (LTL) [Pnu77] is suitable for the encoding of the properties. LTL is a subset of CTL* with modalities referring to time (cf. [HR00,CGP00]). The syntax of the LTL formula is given by the following Backus-Naur-Form (BNF) definition:

$$\varphi ::= \bot \mid \top \mid p \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi\ \mathsf{U}\ \varphi) \mid (\mathsf{G}\ \varphi) \mid (\mathsf{F}\ \varphi) \mid (\mathsf{X}\ \varphi).$$

In addition to the propositional logic operators and predicates p, the temporal operators $\mathsf{X}, \mathsf{F}, \mathsf{G}$ and U are interpreted as follows:

- $\mathsf{X}\ \varphi : \varphi$ must hold at the ne**x**t state.
- $\mathsf{F}\ \varphi : \varphi$ must hold at some **f**uture state.
- $\mathsf{G}\ \varphi : \varphi$ must hold at the current state and all future state (**g**lobally).
- $\psi\ \mathsf{U}\ \varphi : \varphi$ holds at the current or a future state, and ψ must hold up **u**ntil this point. From this point, ψ no longer needs to hold.

LTL formulas are evaluated for a certain path π of states. If we let $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ be a path of states, then π_i is the suffix starting at s_i: $\pi^i = s_i \rightarrow s_{i+1} \rightarrow \ldots$. All temporal logic operators can be related to path expressions, e.g. the ne**x**t-operator's semantics are given by $\pi \models X\phi$ iff $\pi^2 \models \phi$.

Techniques based on Büchi automata have been implemented in SPIN to check if a system meets its specifications. This is done by synthesising an automaton which generates all possible models of the given specification and then checking if the given system refines this most general automaton. SMV and NuSMV employ tableau-based model checking in order to evaluate whether a given LTL formula φ_{LTL} holds. They were originally symbolic model checking tools relying on binary decision diagrams. The set of states satisfying a CTL formula φ_{CTL} is computed as the BDD representation of a fixed point of a function. If all the initial system states are in this set, φ_{CTL} is a system property.

3 PLCs

As already stated, PLCs are a special type of computer based on sensors and actuators able to control, monitor and influence a particular process. There

are many standard tools for the configuration of PLCs, depending on the PLC product family. In these tools, the PLC programming languages standardised in [IEC93] are usually given different names to those in the IEC standard. Thus, the tool used for this study supports, among others, the Statement List (STL) programming language. STL corresponds in its expressiveness one to one to IL, having instructions with the same functionality. The syntax of the two languages differs, however. In the remainder of this paper the designation IL will be used to cover both IL and STL.

3.1 IL Program

An IL program can consist of a number of modules. Each of the modules and the main program contain variable declarations plus a program body. For the purpose of verification, we shall consider the program body as a limited set of lines of code executed in a defined sequence. Let us consider a program $\mathcal{P}$ having $maxpc$ lines of code and n modules $\mathcal{P}_1, \ldots, \mathcal{P}_n$ each having $maxpc_\imath, \imath = 1, \ldots, n$ lines of code. The program $\mathcal{P}$ can then be represented as follows:

$$\mathcal{P} = \{(\jmath, statement_\jmath) \mid \jmath = 1, \ldots, maxpc\} \cup \bigcup_{\imath \leq n} \mathcal{P}_\imath, \text{ where}$$

$$\mathcal{P}_\imath = \{(\jmath_\imath, statement_{\jmath_\imath}) \mid \jmath_\imath = 1, \ldots, maxpc_\imath\}, \text{ for all } \imath = 1, \ldots, n$$

where $statement_\jmath$ ($statement_{\jmath_\imath}$) designates the statement at line $\jmath$ ($\jmath_\imath$) of $\mathcal{P}$ ($\mathcal{P}_\imath$).

3.2 Behavioural Model of an IL Program

The behavioural model of an IL program $\mathcal{P}$ can be represented using a state transition system $\mathcal{T} = (\mathcal{S}, \mathcal{S}_0, \rightarrow)$, where $\mathcal{S}$ is a set of states, $\mathcal{S}_0 \subseteq \mathcal{S}$ a non-empty set of initial states and $\rightarrow$ a transition relation. $\mathcal{S}$, $\mathcal{S}_0$ and $\rightarrow$ are constructed as follows:

Set of states $\mathcal{S}$. The set of states $\mathcal{S} = \mathcal{S}_S \times \mathcal{S}_H \times \mathcal{S}_{PC}$ with

$\mathcal{S}_S$ *- a set of states of software-specific variables.* Software-specific variables are variables defined within the program $\mathcal{P}$. Although IL supports a wide range of data types (relating to its hardware-like nature), in this paper only Booleans and bounded integers are discussed. Let us consider the main program, $\mathcal{P}$, with n_P variables and each module $\mathcal{P}_\imath$ with $n_{P_\imath}$ variables, then $\mathcal{S}_S = \mathcal{S}_{SP} \times \mathcal{S}_{SP_1} \times \ldots \times \mathcal{S}_{SP_n}$ where $\mathcal{S}_{SP} = SP_{var_1} \times \ldots \times SP_{var_{n_P}}$ and $SP_{var_\jmath}$ is a domain of the variable $var_\jmath$ for $\jmath = 1, \ldots, n_P$. For example, for a Boolean variable $var_\jmath$, $SP_{var_\jmath} = \{true, false\}$. The sets $\mathcal{S}_{SP_\imath}$ are defined analogously to $\mathcal{S}_{SP}$.

$\mathcal{S}_H$ *- a set of states of hardware-specific variables.* Depending on the PLC family, various types of CPU register are required for the processing of IL statements. Some of the registers are not important for the verification of IL programs ([PPKE07]). For this reason, only the following registers are considered here: three status bits, two accumulators and a nesting stack (used to save certain items of information before a nesting statement is processed). These registers are represented in the behavioural model of an IL program by hardware-specific variables. The set of states of hardware-specific variables $\mathcal{S}_H$ is constructed as follows. $\mathcal{S}_H = \mathcal{S}_{HP} \times \mathcal{S}_{HP_1} \times \ldots \times \mathcal{S}_{HP_n}$ where the sets in the product correspond to sets of hardware-specific variables of $\mathcal{P}$, $\mathcal{P}_1$, ..., $\mathcal{P}_n$ respectively. Because these sets are equivalent, it is sufficient to define one of them, e.g. $\mathcal{S}_{HP}$. $\mathcal{S}_{HP} = HP_{StatusBits} \times HP_{Accumulators} \times HP_{NestingStack}$ and $HP_{StatusBits} = HP_{RLO} \times HP_{OR} \times HP_{FC}$ (HP_{RLO}, HP_{OR} and HP_{FC} are domains of status bits RLO, OR and FC, that is $\{true, false\}$) and $HP_{Accumulators} = HP_{ACC_1} \times HP_{ACC_2}$ (HP_{ACC_1} and HP_{ACC_2} are accumulator domains). If we let $HP_{NestingStack}$ have l layers, then $HP_{NestingStack} = HP_{Stack_1} \times \ldots \times HP_{Stack_l}$. Each of these stacks contains information to be pushed into the stack before opening a new nesting operation. These are the status bits RLO and OR, and the identifier of the operation before nesting. Thus, $HP_{Stack_\jmath} = HP_{RLO_\jmath} \times HP_{OR_\jmath} \times HP_{Operation_\jmath}$, $\jmath = 1, \ldots, l$, where $HP_{Operation_\jmath}$ is a domain of operation identifiers.

$\mathcal{S}_{PC}$ *- a set of states of program counters.* The program counter of each of the program modules together form the set $\mathcal{S}_{PC}$. Thus, $\mathcal{S}_{PC} = \{1, \ldots, maxpc\} \times \{1, \ldots, maxpc_1\} \times \ldots \times \{1, \ldots, maxpc_n\}$.

Set of initial states $\mathcal{S}_0$. $\mathcal{S}_0 \subseteq \mathcal{S}$, or more precisely $\mathcal{S}_0 = \mathcal{S}_{S_0} \times \mathcal{S}_{H_0} \times \mathcal{S}_{PC_0}$ with $\mathcal{S}_{S_0} \subseteq \mathcal{S}_S$, $\mathcal{S}_{H_0} \subseteq \mathcal{S}_H$ and $\mathcal{S}_{PC_0} \subseteq \mathcal{S}_{PC}$. Sets $\mathcal{S}_{S_0}$ and $\mathcal{S}_S$ may differ merely in the ranges of the variables which can have predifined values for $\mathcal{P}$. Only for this kind of variable can initial values be restricted. For all other software-specific variables, all possible values have to be considered from the very beginning of verification. The initial values of the hardware-specific variables are predefined and identical for each $\mathcal{P}$ and $\mathcal{P}_\imath$, $\imath = 1, \ldots, n$. These variables are initialised with all Booleans being set to $false$ and all integers to 0. Thus, $\mathcal{S}_{H_0}$ is defined by $\mathcal{S}_{H_0} = \prod_{\imath \leq n+1} \mathcal{S}_{HP_0}$, $\mathcal{S}_{HP_0} = HP_{StatusBits_0} \times HP_{Accumulators_0} \times HP_{NestingStack_0}$, $HP_{StatusBits_0} = \{false, false, false\}$, $HP_{Accumulators_0} = \{0, 0\}$ and $HP_{NestingStack_0} = \prod_{\imath \leq l} \{false, false, 0\}$.

Transition relation $\rightarrow$. $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ describes how the state of the model changes after the execution of each statement. These changes are reflected in the fact that new values are assigned to the software and hardware variables, and the program counter. After each statement, the program counter is given a

new value, pointing to the next statement to be executed. Only one software-specific variable, which at the same time is the statement argument, can be changed by a single statement. On the other hand, one statement can change a number of hardware-specific variables. For more details of, how the transition relation and behavioural model are constructed, see [PPKE07].

4 Automated Transformation of IL Programs

The automated transformation of IL programs described in [PPKE07] was developed as a part of a masters thesis [Fen07]. As shown in Fig.1, besides the program to be transformed, the software also needs a description of the IL language. This description is supplied in an IL metafile (cf. Fig.2). The result of the automated transformation of the IL program is a corresponding NuSMV model. In some cases it is possible to reduce the state space of the resulting NuSMV model by manual optimisation. More details of the above steps are given in the sections below. On the basis of the NuSMV model and specification being proven, the next step in the verification is performed by the NuSMV model checker.

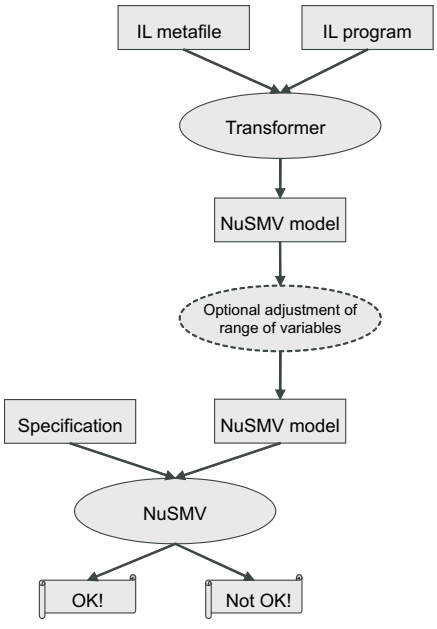

Fig. 1. Transformation process

4.1 Metafile

The metadescription of the IL language is defined in a simple text file with a special format (metafile). Part of the contents of the IL metafile is shown in Fig.2. This simple description of the language makes the software more universal and allows us to use it for the transformation of programs written in a number of other low-level languages as well, provided a metadescription of the language.

As shown in Fig.2, the system hardware variables must be described at the beginning of the metadescription. This is done using *identifier, type, initial value* triples. For example, *RLO,0,0* means that the status bit *RLO* is a Boolean (type 0) and has the initial value 0, and *ACC1,1,0* means that *ACC*1 is an integer (type 1) with the initial value 0. In the second part of the metafile, the IL statements are described. There are four types: 0-statements - with an argument, 1-statements - with a nesting operation, 2-statements - with an effect on the program counter, and 3-statements - with no argument. Accordingly, the conjunction *A argument* is of type 0 and described by

$$A,0,if(FC=1)\{RLO:=OR||RLO\&\&\_ARG\_);\}$$
$$else\{RLO:=\_ARG\_;FC:=1;\}$$

This means: if *FC* is true, the *RLO* bit is set to *OR ||(RLO && argument)*, otherwise *RLO* is set to *argument* and *FC* is set to *true*.

```
[variables]
RLO,0,0
OR,0,0
FC,0,0
ACC1,1,0
ACC2,1,0
...[meta]
A,0,if(FC=1){RLO:=OR||(RLO&&_ARG_);}else{RLO:=_ARG_;FC:=1;}
JU,2,PC:=_ARG_;
+I,3,ACC1:=ACC2+ACC1;
*I,3,ACC1:=ACC2*ACC1;
>I,3,if(ACC2>ACC1){RLO:=1;OR:=0;FC:=1;}else{RLO:=0;OR:=0;FC:=1;}
<I,3,if(ACC2<ACC1){RLO:=1;OR:=0;FC:=1;}else{RLO:=0;OR:=0;FC:=1;}
...
```

Fig. 2. Metadescription of the IL language

4.2 Manual Optimisation of the NuSMV Model

In some cases the NuSMV model resulting from the transformation of the IL program will have an optimisation facility. The model optimisation is optional and has to be performed manually. An illustration of this will now be given. Let

us consider some integer variables in an IL program having a restricted range of integer values. Despite the limitation of the variables, a whole range of integers is reserved for them. These variables do not need the entire range of integers in the corresponding NuSMV model, and problems may result due the excessive size of the model's state space. Provided the ranges of the variables are known, they can be adjusted accordingly and the space requirement reduced.

5 Case Study

This section takes a closer look at the process of formal verification of IL programs already described. An IL program and the corresponding NuSMV model are presented. To show the behavioural equivalence between the IL program and its NuSMV model, the method proposed in [PH07] can be applied. The most complex issue in the verification process turns out to be the implementation of a function call. We have therefore chosen to demonstrate how this is done on the basis of a sample IL program. For more about IL programming [Gie03] and [Sie04] should be consulted.

5.1 IL Program

An outline of the program considered here (*DemonstrateFormByte*) is shown in Fig.3. This simple IL program demonstrates the call of the function *FormByte* which takes 8 bits as input (*Bit0, Bit1,...*, *Bit7*) and combines them into one byte (*Byte*).

5.2 NuSMV Model

A NuSMV program consists of several modules. There must be one module with the name *main* and no formal parameters ([CCB+02]). Accordingly, the IL program is implemented by the main module in NuSMV and the function called by a further module, which is instantiated in the main module (cf. Fig.4). For more information about how IL statements are transformed into NuSMV model, see [PPKE07].

Let us consider the transitions given in Fig.4. The program *DemonstrateFormByte* has 2 lines, in the first of which the function *FormByte* is called. This call is implemented in the NuSMV model by saying in the main module that if program counter is equal to 1 and *FormByte* has not finished executing, the program counter of the main module does not change its value. Only when *FormByte* has finished its execution may the main module program counter increment its value.

```
//Program "DemonstrateFormByte"
//8 boolean and 1 integer variables
//program body
CALL "FormByte" (
   InputBit0 := Bit0, InputBit1 := Bit1, InputBit2 := Bit2,
   InputBit3 := Bit3, InputBit4 := Bit4, InputBit5 := Bit5,
   InputBit6 := Bit6, InputBit7 := Bit7, OutputByte := Byte);
//Function "FormByte"
//input: 8 booleans (InputBit0,...,InputBit7)
//output: 1 integer (OutputByte)
//1 temporary variable (Value) which keeps temporary result
//function body
   L 0; //Initialise the temporary result
   T Value; //Value=0
   AN InputBit0; //Check if Bit0 is set
   JC BIT1; //if not jump to BIT1
   L 1; //Adjust the temporary result by 2**0
   T Value; //Value=1
BIT1: AN InputBit1; //Check if Bit1 is set
   JC BIT2; //if not jump to BIT2
   L Value; //Adjust the temporary result by 2**1
   L 2;
   +I ;
   T Value; //Value=Value+2
   ...
```

Fig. 3. An outline of the IL program demonstrating the calling of the function which combines the eight bits supplied into one byte

```
MODULE FormByte(param0,param1,param2,param3,param4,param5,param6,param7)
...
MODULE main
//...variable declaration
//the instantiation of the FormByte module is realised in the next
line
CALL_FormByte : FormByte(Bit0,Bit1,Bit2,Bit3,Bit4,Bit5,Bit6,Bit7);
ASSIGN
next(PC) :=
   case
     PC=1 & CALL_FormByte.PC<63: 1;
     PC=1 & CALL_FormByte.PC=63: 2;
     1 : PC;
   esac;
init(PC) := 1;
...
```

Fig. 4. An outline of the NuSMV model corresponding to the IL program DemonstrateFormByte

5.3 Specification and Verification Results

To prove the correctness of the NuSMV model we need to check if the byte value obtained corresponds to the eight bits supplied. This property can be represented by the following LTL formula:

$$\mathsf{G}(PC = 2 \Rightarrow Byte = (Bit0 + 2 * Bit1 + 4 * Bit2 + 8 * Bit3 + 16 * Bit4 + 32 * Bit5 + 64 * Bit6 + 128 * Bit7))$$

Unfortunately, proving this property by the method described is inefficient. It took over eight hours to do so. The ultimate aim of this work study, however, is to apply the proving technique to far more complex case studies. Hence, the approach needed to be improved. How this was done, is described in the next section.

5.4 Improvement of the Method

The reason for the inefficiency of the verification lay in the enormous number of transitions which had to be considered by the NuSMV model checker when instantiating a new module in a main module. More precisely, all the variables that formed the state space of the module *FormByte* ware also part of the state space of the main module. These variables are, however, of no importance before and after the module *FormByte* is referenced in the main module (hereafter this situation will to be referred to as *module FormByte is not active*).

Considering the above, a constraint was required in the main module with the meaning: "if the *FormByte* module is not active, the model checker only checks the states in which the *FormByte* variables are set to their initial values". This could be achieved by means of the following invariant:

```
INVAR (PC!=1 -> (CALL_FormByte.PC=1  |  CALL_FormByte.PC=64) &
CALL_FormByte.Bit0=0 & CALL_FormByte.Bit1=0 & CALL_FormByte.Bit2=0 &
CALL_FormByte.Bit3=0 & CALL_FormByte.Bit4=0 & CALL_FormByte.Bit5=0 &
CALL_FormByte.Bit6=0 & CALL_FormByte.Bit7=0 & CALL_FormByte.Byte=0 &
CALL_FormByte.Value=0 & CALL_FormByte.RLO=0 & CALL_FormByte.OR=0 &
CALL_FormByte.ACC1=0 & CALL_FormByte.ACC2=0)
```

Besides the addition of this invariant to the main module, some changes to *FormByte* were necessery. In order to enable the *FormByte* variables to have their initial values when the module was inactive, some new transitions had to be added. These transitions needed to set the variables to the predefined values once execution of the *FormByte* function had terminated. For this, the *FormByte* program counter was incremented by 1. Additionally, for each variable a new transition was formed, which set the variable to its initial value.

By adding this invariant to the model we succeeded proving the relevant property in 113.8 seconds, a vast improvement on the previous result. Thus the changes described brought about a marked improvement in our method.

6 Conclusion and Future Work

The safety demands of many systems based on PLC are considerable. The formal verification of the PLC software is thus of great importance. The verification method demonstrated in this paper is a powerful instrument for analysing safety-related software.

An approach was presented for the automated transformation of IL programs into NuSMV models. This is supported by a tool, which was also described. The efficiency and convenience of the tool were demonstrated by means of a case study. Because the transformation can be automated, the approach has the potential for a wide application in industry.

Although the approach is stable there is, however, still scope for improvement. While the tool was being developed aspects for optimisation were identified and appropriate features implemented directly; others are due to be incorporated in the near future (the invariant described in the previous section, for example, is to be automatically added to the NuSMV model). The aim of the project is to efficiently verify PLC software of as high a complexity as possible. In order to achieve this goal we will need to continue refining the technique. Thus, the development of the approach itself and the accompanying tool are ongoing. Further work in this field should provide a suitable set of reduction methods which will allow the state explosion problem, arising from the growing model size, to be resolved.

References

[CCB+02] Roberto Cavada, Alessandro Cimatti, Marco Benedetti, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Roberto Sebastiani. NuSMV: a new symbolic model checker. http://nusmv.itc.it/, 2002.

[CCL+00] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen. Towards the automatic verification of PLC programs written in Instruction List. In *Proc. IEEE Int. Conf. Systems, Man and Cybernetics (SMC'2000), Nashville, TN, USA, Oct. 2000*, pages 2449–2454, 2000.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doran A. Peled. *Model Checking*. MIT Press, 2000.

[Fen07] G. Fendoglu. Überführung eines AWL-Modells in ein NuSMV-Modell. Masters thesis. Technische Universität Braunschweig, 2007.

[Fig06] C. Figura. Überdeckungstests für fehlersichere Funktionspläne auf Basis einer geeigneten Überführung. Master thesis. Martin-Luther-Universität, 2006.

[Gie03] Walter Giessler. *SIMATIC S7 SPS-Einsatzprojektierung und -programmierung*. VDE Verlag GMBH, 2003.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. In *IEEE Transactions on Software Engineering*, volume 23, pages 279–295, 1997.

[HR00] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science - Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[IEC93] IEC. *International Electrotechnical Commission Standard 61131-3, Programmable controllers - Part 3*, 1993.

[McM96] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, second edition, 1996.

[PH07] J. Peleska and E. Haxthausen. Object Code Verification for Safety-Critical Railway Control Systems. In E. Schnieder and G. Tarnai, editors, *Proc. of the 6th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)*. GZVB, 2007.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[PPKE07] O. Pavlovic, R. Pinger, M. Kollmann, and H. D. Ehrich. Principles of Formal Verification of Interlocking Software. In E. Schnieder and G. Tarnai, editors, *Proc. of the 6th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)*. GZVB, 2007.

[Sie04] Siemens. *SIMATIC Anweisungsliste (AWL) fuer S7-300/400. Referenzhandbuch (SIMATIC Instruction List for S7-300/400. Reference Manual)*, 2004.