VERIFICATION OF PLC PROGRAMS USING FORMAL PROOF TECHNIQUES

Andre Sülflow, Rolf Drechsler
Graduate School Embedded Systems (GESy)
University of Bremen
Address: Bibliothekstraße 1, D-28359 Bremen, Germany
Phone: +49-421-218-63945, Fax: +49-421-218-7385,
E-Mail: {suelflow,drechsle}@informatik.uni-bremen.de

Abstract: The application of Programmable Logic Controllers (PLCs) in safety critical systems demands a failure free behavior considering all possible scenarios. Due to the cost of software development a user program is often in use on different types of PLCs. But one open question is: Behaves the user program equivalent on all PLCs?
We propose a framework suitable to prove the equivalence of a user program regarding different types of PLCs. The semantic behavior of the hardware is embedded and the equivalence against a high level reference model is proved. We apply formal *Equivalence Checking* (EC) using *Boolean satisfiability* (SAT) and evaluate the framework in a case study.
Keywords: formal verification, instruction list, SAT, equivalence checking

1. INTRODUCTION

Programmable Logic Controllers (PLCs) are widely used in todays industry. A PLC is a reprogrammable computer, based on sensors and actors, which is controlled by a user program. They are highly configurable and thus are applied to various industrial sectors like e.g. automotive, chemical devision or rail automation systems.

To ensure the correct behavior of a user program one can use test and simulation approaches, or formal methods, that provide a proof of correctness, i.e. testing the *complete* search space. The focus of this work is on formal verification.

Today, one problem is the verification of software running on PLCs. A user program is executed on different types of PLCs (hardware), but an open question is: Is the software running equivalent on all systems? Our focus is the proof of equivalence. We present an approach to transform a user program to a high level implementation by embedding the semantic of the underlying hardware. Afterwards the implementation is formally proved to be equivalent to a reference model. That allows e.g. an estimation of the influence of hardware specific parts to the behavior of a user program.

The *International Electrotechnical Commission* (IEC) standardized two textual and three graphical programming languages for PLCs (IEC, 1993): *Structured Text* (ST), *Instruction List* (IL) and *Ladder Diagram* (LD), *Function Block Diagram* (FBD), *Sequential Function Chart* (SFC). We focus on IL, that is widely applied for programming todays PLCs.

In (Canet *et al.*, 2000; Pavlovic *et al.*, 2007*b*) the principles of formal verification of IL programs were proposed and evaluated in (Pavlovic *et al.*, 2007*a*). These works present an overview of how IL statements can be translated to a formal language. The focus is on symbolic model checking (McMillan, 1993) of a specification given in *Linear Temporal Logic* (LTL) using NuSMV (Cimatti *et al.*, 2002) as core solver. (Peleska and Haxthausen, 2007) applied a transformation technique to show the equivalence of two PLC programs. In contrast we focus on formal *Equivalence Checking* (EC) using todays state-of-the-art *Boolean satisfiability* (SAT) solvers (Davis and Putnam, 1960; Eén and Sörensson, 2004).

Our contribution is (1) to provide a fully automatic method for translating IL programs to a higher level model in *SystemC* and (2) to apply formal *Equivalence Checking* using SAT as the underlying technique. The benefits of *SystemC* as an intermediate representation are (1) the simulation facility within the *SystemC* environment and (2) formal verification can be applied (Drechsler *et al.*, 2005). An overview of the proposed framework is given in *Figure 1*.

In a case study complex programs are considered. The results show that todays SAT solvers can handle even large models within a short run time.

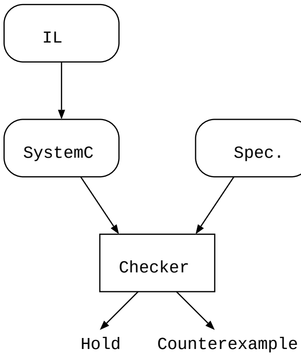

Fig. 1. Framework overview

The work is structured as follows: In Section 2 the preliminaries for PLC programming, *SystemC* and SAT are presented, followed by a detailed introduction of the conversion of IL programs to *SystemC* in Section 3. Section 4 focuses on *Equivalence Checking* using SAT. The experimental results are presented in Section 5.

2. PRELIMINARIES

To keep this paper self-contained this section gives an overview of the languages and techniques used in the remaining sections. The section starts with a description of the considered PLC, continues with PLC and IL programming and ends with an introduction to *SystemC* and SAT.

2.1 PLC

The focus is on a class of PLCs that repeats the execution of a user program periodically. That are e.g. PLCs capable for railway electronic interlocking specified to *Safety Integrity Level 3* (SIL3) by the *International Electrotechnical Commission* (IEC) in (IEC, 1998). The task for the operating system is to control the connected peripherals and to provide interfaces for reading and writing of data by a user program.

The three main phases for program execution are as follows:

1. Read from inputs (sensors)

2. Program execution

3. Write to outputs (actors)

Therefore during program execution the inputs are assumed to be "stable". After execution the computed data for the outputs is written to the connected modules. The next cycle starts with reading from inputs.

2.2 PLC Program

The entry point of a PLC user program is a "main" module. The main module itself can be structured into submodules. Each of the modules provides an interface, i.e. inputs and outputs. Therefore the program code can be partitioned into several levels and thus can be re-used. If a module needs temporary or static variables then a data block is assigned. Two or more modules can share one data block (Siemens, 2001).

There are two types of modules: (1) without and (2) with static variables. The first one is called a combinatorial module, whereas the second one is referred as a sequential module. In combinational modules there are only temporary variables allowed. That is, each temporary variable is reset after program execution and all computed data are lost. A sequential module keeps the current values of static variables for one of the following cycles. For example, a counter is a sequential module.

2.3 IL Program

IL is one of the textual PLC programming languages. It is a low-level, assembler-like language. A module in IL consists of two sections: (1) variable declaration and (2) a set of instructions in a program body. In the reminder of the paper, IL programs written in *Statement List* (STL) are considered. STL is semantically equivalent to IL, but the instructions have a different syntax (Siemens, 2003).

In the declaration section all variables, that are referenced in the definition part, are declared. This includes interface variables, i.e. IN, OUT, INOUT, as well as temporary (TEMP) and static (STAT) variables. Each variable has a type e.g. INT (16-bit) or BOOL (1-bit).

An IL program consist of n lines of code. Each line contains one instruction, i.e. one operator, that has at most one operand. The instructions are sequentially executed in a deterministic order. The control flow is influenced by e.g. a jump instruction, referencing a label, that marks a line k, $1 \leq k \leq n$. If the jump is executed the program continues execution at line k.

2.4 SystemC

SystemC is an open-source C++ class library and can be downloaded free of charge (Synopsys Inc. and CoWare Inc. and Frontier Design Inc., 2008). *SystemC* is platform independent and can be used on every system with a standard C++ compiler available, e.g. Windows, Linux, Unix and Mac OS X. In the year 2005 the *SystemC* library was standardized in IEEE 1666-2005 (IEEE, 2005).

During the last years the *SystemC* community has grown. Today *SystemC* is applied and integrated in the development flow on academic as well as industry sites. *SystemC* provides an cycle accurate simulator and systems can be modeled at different abstraction levels - from software down to the hardware description level. Additionally, each module can be partitioned into several submodules. That makes *SystemC* a flexible framework suitable to e.g. simulate PLC programs.

2.5 SAT

The *Boolean satisfiability problem* (SAT problem) is to check for a given Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ whether there exists an assignment that evaluates f to 1 or not. In general the function f is given in *Conjunctive Normal Form* (CNF). Each CNF is a set of clauses, where each clause is a set of literals and each literal is a propositional variable or its negation. Each logic operator as well as arithmetic and relation ones are translated to a set of clauses.

In the recent years several improvements in SAT were achieved. Thus, todays state-of-the-art SAT solvers can handle instances with hundreds of thousands of variables and clauses. This enables the application of SAT for a wide range of formal verification problems.

3. SYSTEMC MODEL

To obtain a model for formal verification, *SystemC* is used as intermediate, high level representation of an IL program. In the following we make use of the integrated *SystemC* environment *SyCE* (Drechsler *et al.*, 2005). That allows to e.g. synthesize (Fey *et al.*, 2004) and formally verify (Große and Drechsler, 2005) *SystemC* designs using SAT.

In this section details of the semantic equivalent transformation of an IL program to *SystemC* are presented. We start with the underlying *Central Processing Unit* (CPU) (Section 3.1), followed by the transformation of variables (Section 3.2), basic statements (Section 3.3), control flow statements (Section 3.4) and call statements (Section 3.5).

3.1 CPU

An IL program is executed on a CPU. Therefore to obtain a semantic equivalent model of an IL program the semantic behavior of the CPU has to be considered and implemented, too. In this work the CPU model of (Siemens, 2003), the same as the one in (Pavlovic *et al.*, 2007*b*), is used. An overview of the main registers is given, for more details it is referenced to (Siemens, 2003).

The CPU consists of several registers, representing its current state: *accumulators*, *status word*, *nesting stack* and the *master control relay stack*.

Depending on the chosen CPU type, a CPU may have two or four accumulators. The width of each accumulator is 32-bit. Accumulators are used e.g. to perform arithmetic operations on two integer variables. The IL programs in the case study use the functionality of two accumulators only, therefore a CPU with two accumulators is used as reference.

The *status word* has a bit width of 16-bit, but only the first 9 bits are used: /FC (first check), RLO (result of logic operation), STA (status), OR (temporary result of an *or* operation), OS (overflow stored), OV (overflow), CC0 (condition code 0), CC1 (condition code 1) and BR (binary result) (Siemens, 2003). All status bits are considered and implemented.

The *nesting stack* is used to store intermediate results. In detail, it saves the current value of BR, OR, RLO and an additional *operation identifier* (OI) on a stack. A maximum of seven stack elements are supported. Thus, the *nesting stack* offers facilities to build more complex functions.

The status of a software *master control relay* (MCR) influences the semantic of some IL instructions. If the MCR is enabled, the instructions behave normal, otherwise the result is independent from the values of the CPU registers. The values of the *MCR stack*, consisting of eight elements, one bit each, controls the MCR. The MCR was not considered in the models of our case study, therefore we prevented the implementation.

3.2 Variables

For all IL data types, e.g. *BOOL* and *INT*, there exists an equivalent in the *SystemC* syntax. For example, an *INT* has a bit width of 16 and is syntactically equivalent to *sc_int<16>* in *SystemC*. Therefore in general the declaration of variables of an IL program is mapped one-to-one to *SystemC* (see *Table 1*).

But for input and output variables the semantic behavior is different. In the following the differencing behavior is described in detail and a solution is provided.

IL	SystemC	
Type	Type	Behavior
IN	sc_in<T>	signal
OUT	sc_out<T>	signal
INOUT	sc_inout<T>	signal
TEMP	function variable	variable
STATIC	member variable	variable

Table 1. Mapping of data types

The first column of *Table 1* gives the IL data type, followed by the corresponding *SystemC* data type in column two. The last column represents the behavior of the data types in *SystemC*. The parameter *T* defines a data type, e.g. *sc_in<sc_int<16> >* is an 16-bit integer input signal.

SystemC provides a cycle accurate simulator. Therefore it is distinguishing between the behavior of a signal and a standard variable. A signal has in one cycle exactly one value assigned, whereas a standard variable can have more than one value. In contrast an IL variable behaves like a variable in *SystemC*.

	Behavior	
Line	Variable	Signal
1	$int\ A$;	$sc\_inout < int > A$;
2	A_n := A_c + 1;	A_n := A_c + 1;
3	A_n := A_n + 2;	A_n := A_c + 2;

Table 2. *SystemC* behavior

Example 1. *Consider Table 2, representing a code fragment that shows the difference between a variable and a signal behavior. The columns give the line number (Line) and the behavior type (Variable, Signal). The current and next state of A are represented by A_c and A_n.*

This simple program performs an addition of the values 1 and 2 to an integer variable A. First, A is declared (Line 1). Then, the value 1 is added to the initial value A_c of A (Line 2). The intermediate result A_n is equal in both cases. Afterwards the difference is observed (Line 3): A read operation on a signal results in returning the value of A_c, whereas a read on a variable returns the last assigned value A_n.

Line	Code
1	$sc\_inout < int > A\_inout$;
2	$int\ A_c$:= A_inout_c;
3	A_n := A_c + 1;
4	A_n := A_n + 2;
5	A_inout_n := A_n;

Table 3. IL conform behavior

To solve the differencing behavior temporary variables are used for all variables of type IN, OUT and INOUT (see *Table 3*). First, the value of the original variable is copied to a local variable (Line 2). The following operations are then performed on the local variable only (Lines 3-4). At the end of the program the value of the local variable is copied to the original variable (Line 5). Therefore the semantic equivalent behavior to an IL variable is reached.

3.3 Basic statements

The existing documentation (Siemens, 2003) of the semantic of the underlying CPU makes the translation to *SystemC* straight-forward. An IL program consists of n lines of code with exactly one instruction per line. An instruction consists of one operator and in general zero or one operand[1].

For each instruction there exists a corresponding transformation to *SystemC* that can be applied in an automatic process. In *Table 4* a few transformation rules are presented. On the left side the IL statement with the operator (code), the parameter type of the operand (*op*) and a short description is given (description). The right side represents the corresponding code fragment that is to be replaced in *SystemC*.

Example 2. *Consider the IL instruction "A ctrl", with ctrl : BOOL. The operator "A" is equivalent to the logical "and", and the transformation rule is obtained from Table 4. Therefore the instruction above is translated to: RLO = (/FC)? (ctrl | OR) & RLO : (ctrl | OR); /FC=1;*

3.4 Control flow statements

Control flow statements, i.e. jumps, can be handled in several ways in *SystemC*. The focus is in the following on two solutions: (1) an explicit modeled program counter (PC) (see e.g (Pavlovic *et al.*, 2007*b*)) and (2) goto statements. The IL code fragment in *Figure 2* is considered and the focus is on the PC behavior.

[1]Except for the CALL statement, that can have more than one operand.

IL statement			SystemC
code	*op*	description	
Bit op.			

A *op*	bit	logical and	RLO = (/FC)?(*op* | OR) & RLO : (*op* | OR);
			/FC = 1;

O *op*	bit	logical or	RLO = (/FC)? *op* | RLO : *op*;
			OR = 0; /FC = 1;

= *op*	bit	assigns RLO	*op* = RLO; OR = 0; /FC = 0;
Transfer op.			
L *op*	byte, word, dword	loads data	ACCU2 = ACCU1; ACCU1 = *op*;
T *op*	byte, word, dword	stores data	*op* = ACCU1;
Arithmetic op.[a]			
+I		addition	ACCU1 = ACCU2 + ACCU1;
*I		multiplication	ACCU1 = ACCU2 * ACCU1;
Relation op.[b]			

==I		equal	RLO = (ACCU2 == ACCU1);
			OR = 0; /FC = 1; OV = 0;

[a]OS, OV, CC1 and CC0 are changed depending on the computed result (see (Siemens, 2003)).

[b]CC1 and CC0 are changed depending on the compare (see (Siemens, 2003)).

Table 4. Statement translation

```
        AN      #bit0;
        JC      BIT1;
        =       #var;
BIT1:   NOP     0;
```

Fig. 2. Example program

Explicit modeling of a PC can be implemented by a Finite State Machine (FSM) (see *Figure 3*). The FSM is controlled by the current value of the PC counter. Thus, for each line of IL code there exists a case statement. A jump statement changes the PC counter to the jump target, if the jump has to be executed and increments the PC otherwise (Line 7).

```
1  int PC = 1;
2  while (PC <= 4) {
3    switch (PC) {
4    case 1: //AN #bit0;
5    PC = PC + 1; break;
6    case 2: //JC  BIT1;
7    PC = RLO?4:PC + 1; break;
8    case 3:    //= #var;
9    PC = PC + 1; break;
10   case 4: //BIT1: NOP 0;
11   PC = PC + 1; break;
12   }
13 }
```

Fig. 3. FSM model

This mapping has the disadvantage that an extra variable for the PC is needed which makes the "traceability" harder. The FSM is optimized (see *Figure 4*) by using case statements only for $PC = 1$ (Line 5), the start of the program, and each line marked with a label (Line 10).

```
1  int PC = 1;
2  bool stop = false;
3  while (!stop) {
4    switch (PC) {
5    case 1:
6    //AN #bit0;
7    //JC BIT1;
8    if (RLO) {PC = 4; break;}
9    //= #var1;
10   case 4: //BIT1: NOP 0;
11   stop = true;
12   }
13 }
```

Fig. 4. Optimized FSM model

The second option is to use the C++ *goto* statement. Each line that contains a label in IL, gets the same label in C++. Therefore a jump will be executed by calling *goto*. The advantage is that the original names of the labels will be kept from IL. Therefore it makes a potential debugging easier.

3.5 Calls

The IL *CALL* statement executes a submodule. A call gets a list of IN, OUT, and INOUT parameters. *CALLs* are implemented by creating a C++ function for each submodule and passing the parameters by copy (IN) or reference (OUT, INOUT).

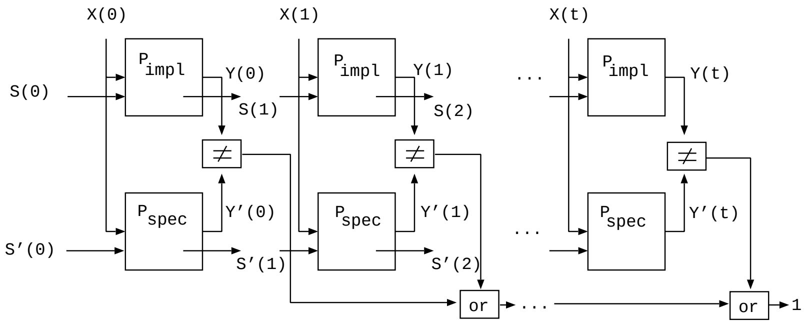

Fig. 5. Sequential EC with reset states

4. EQUIVALENCE CHECKING

To formally verify the equivalence of a given implementation P_{impl} and its specification P_{spec}, SAT-based *Equivalence Checking* (EC) is applied. First, both models are assumed to have the same number of IN, OUT, INOUT variables, otherwise the models are declared as non-equivalent.

The implementation as well as the specification are given in *SystemC*. By using a *SystemC* parser, e.g. (Fey *et al.*, 2004), the models are analyzed and translated into a SAT instance. In the following the SAT formulation is presented for combinatorial and sequential models.

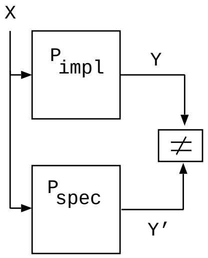

Fig. 6. Combinational EC

4.1 Combinatorial

For combinatorial models there are inputs (X), outputs (Y) and temporary variables. The equivalence of P_{impl} and P_{spec} is proved by creating a SAT instance as follows (see Figure 6): (1) create both models, (2) connect the inputs and (3) force the outputs to be different. A name based matching algorithm is used to find the corresponding variable $v \in \{X, Y\}$ of P_{impl} in P_{spec}. If the instance is unsatisfiable the models are equivalent, otherwise it is satisfiable, the models are different, and a counterexample is provided.

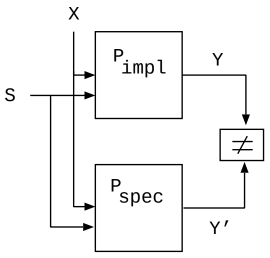

Fig. 7. Sequential EC without reset states

4.2 Sequential

Additional to the inputs (X), outputs (Y) and temporary variables there are state variables (S). States keep the value of the current cycle and are available in the next cycle.

A general sequential EC model is shown in *Figure 5* (Pixley, 1992). As in bounded model checking (Biere *et al.*, 1999) both models are unrolled for t time frames; the inputs (X) are connected and the outputs (Y) are forced to be different in at least one of the time frames. In the worst case an unrolling (parameter t) has to be done until the sequential depth is reached[2]. For the initial state values (S(0), S'(0)) two cases are distinguish:

[2]The sequential depth is longest trace without repeating a state. It is also known as recurring diameter.

(1) with and (2) without given reset states.

With reset states additional constraints are added in the SAT instance, which force S(0) and S'(0) to be identical to the reset values.

For PLCs the variables are initialized with zero or false (Pavlovic *et al.*, 2007*a*). Considering reset states leads in the worst case to an unrolling to the sequential depth. Therefore this approach can be time and memory consuming.

Thus, we concentrate on the case without explicitly given reset states. The problem is reduced to one time frame by using a name-based state matching, i.e. find and connect the equivalent initial states in P_{impl} and P_{spec}. In *Figure 7* an abstract representation of the SAT instance is shown. One time frame is created, the inputs (X) are connected and the outputs (Y) are forced to be different. Additional constraints are added to the SAT instance that match and connect the state variables (S) of both models. Therefore, as for the inputs, assigning an initial value in P_{impl} leads to the same assignment in P_{spec}. Note, letting S uninitialized is an over-approximation of the state space and may lead to *false negatives*, i.e. implementation and specification are wrongly declared to be not equivalent.

5. EXPERIMENTAL RESULTS

In the experimental study combinatorial and sequential models, used in the railway electronic interlocking domain, are considered.

	Variables					IL
Program	I	O	I/O	T	S	LOC
Comb. Model						
comb-1	8	1	-	1	-	63
comb-2	8	1	15	2	-	52
comb-3	1	-	2	-	-	22
Seq. Model						
seq-1	23	10	2	25	70	775

Table 5. IL models

Details of the IL models are presented in Table 5. It contains the number of inputs (I), outputs (O), in/outputs (I/O), temporary variables (T), number of state variables (S) and the number of IL code lines in the body of the program (LOC).

Sequential EC without explicit reset states is applied, i.e. proving the equivalence for all possible initial states and using a state matching. The high level specification was manually created and implemented in *SystemC*.

All experiments have been carried out on an Intel Core 2 Duo processor (2.33 GHz, 2 GB main memory, Mac OS X 10.4) using *MiniSat* (Eén and Sörensson, 2004) as underlying SAT solver.

Program	Impl. #ops	Spec. #ops	Time
Comb. Model			
comb-1	120	41	0.01s
comb-2	498	79	0.03s
comb-3	326	14	0.06s
Seq. Model			
seq-1	18330	1128	3.79s

Table 6. Equivalence Check

In Table 6 the name of the model (Program), the number of operators[3] in the implementation (Impl. #ops) and the specification (Spec. #ops) are shown. The run time of the SAT solver is presented in the last column (Time).

Due to the extra hardware specific variables, i.e. registers, accumulators and *nesting stack*, an IL program has a factor of 3 to 23 more operations than its higher level specification. The factor depends on the type of instruction in the IL program. For example, an instruction using the *nesting stack* needs more operators than one without.

Regarding run time SAT-based EC is fast. For the combinatorial models (comb-1, comb-2, comb-3) the run time is near to zero, but also for the sequential case the equivalence was proved within four seconds only.

Therefore SAT-based EC was shown as a powerful technique to cope with formal verification of PLC programs.

6. CONCLUSION

A framework that automatically embeds IL programs and the semantic of PLC specific hardware into a *SystemC* model was proposed. The equivalence to a reference model was proved by using SAT-based *Equivalence Checking*. It was shown, that modern SAT solver can formal verify IL programs within a short run time.

For further work the extension of the supported subset of IL instructions, e.g. considering shift and timer operations, is planned. Additionally, the appliance to formal *Property Checking* using SAT is in focus.

ACKNOWLEDGEMENT

This work was supported in part by the *Rail Automation Graduate School (RA:GS!)* of Siemens Transportation Systems in Braunschweig, Germany.

[3]An operator is e.g. an assignment, arithmetic or relation operation.

REFERENCES

Biere, A., A. Cimatti, E. Clarke and Y. Zhu (1999). Symbolic model checking without BDDs. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 1579 of *LNCS*. Springer Verlag. pp. 193–207.

Canet, G., S. Couffin, J.-J. Lesage, A .Petit and P. Schnoebelen (2000). Towards the automatic verificication of PLC programs written in instruction list. In: *IEEE conf. on Systems, Man and Cybernetics (SMC)*. pp. 2449–2454.

Cimatti, A., E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella (2002). NuSMV 2: An OpenSource tool for symbolic model checking. In: *Computer Aided Verification*. pp. 359–364.

Davis, M. and H. Putnam (1960). A computing procedure for quantification theory. *Journal of the ACM* 7, 506–521.

Drechsler, R., G. Fey, C. Genz and D. Große (2005). SyCE: An integrated environment for system design in SystemC. In: *IEEE International Workshop on Rapid System Prototyping*. pp. 258–260.

Eén, N. and N. Sörensson (2004). An extensible SAT solver. In: *SAT 2003*. Vol. 2919 of *LNCS*. pp. 502–518.

Fey, G., D. Große, T. Cassens, C. Genz, T. Warode and R. Drechsler (2004). ParSyC: An Efficient SystemC Parser. In: *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*. pp. 148–154.

Große, D. and R. Drechsler (2005). *CheckSyC*: An efficient property checker for RTL SystemC designs. In: *IEEE International Symposium on Circuits and Systems*. pp. 4167–4170.

IEC (1993). *IEC-61131-3: Programmable controllers - Part 3: Programming languages*.

IEC (1998). *IEC-61508: Functional safety of electrical/electronic/programmable electronic safety-related systems*.

IEEE (2005). *IEEE-1666: IEEE Standard SystemC Language Reference Manual*.

McMillan, K.L. (1993). *Symbolic Model Checking*. Kluwer Academic Publisher.

Pavlovic, O., R. Pinger and M. Kollmann (2007*a*). Automated formal verificaiton of PLC programs written in IL. In: *Conference on Automated Deduction (CADE)*. pp. 152–163.

Pavlovic, O., R. Pinger, M. Kollmann and H.-D. Ehrich (2007*b*). Principles of formal verification of interlocking software. In: *Proc. of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT)*. pp. 370–378.

Peleska, J. and A.E. Haxthausen (2007). Object code verification for safety-critical railway control systems. In: *Proc. of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/-FORMAT)*. pp. 184–199.

Pixley, C. (1992). A theory and implementation of sequential hardware equivalence. *IEEE Trans. on CAD* 11(12), 1469–1478.

Siemens (2001). *Grundlagen zur SPS-Programmierung mit SIMATIC S7-300*.

Siemens (2003). *SIMATIC–Statement List (STL) S7-300 and S7-400 Programming*.

Synopsys Inc. and CoWare Inc. and Frontier Design Inc. (2008). *Open SystemC Inititative*. http://www.systemc.org.