Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

Improving large-sized PLC programs verification using abstractions ⋆

V. Gourcuff * **O. de Smet** * **J.-M. Faure** *

* *LURPA, ENS de Cachan*
61, avenue du Président Wilson
94 235 Cachan cedex - FRANCE
{*gourcuff,de_smet,faure*}*@lurpa.ens-cachan.fr*

Abstract: This paper proposes a formal representation of logic controllers programs that is aiming at improving scalability of model-checking techniques, when verifying controllers extrinsic properties. This representation includes only the states which are meaningful for properties proof and minimizes the number of variables that feature each state. Comparison with previously proposed representations, on the basis of three increasing complexity examples, validates this representation and quantifies its efficiency.

1. INTRODUCTION

In order to improve dependability and safety of critical automated systems, several industrial standards address the development of the Programmable Logic Controllers (PLCs) which are included in these systems. We can quote in particular the IEC 61508 standard (IEC [2000]) that deals with functional safety. This standard recommends the use of formal methods for the programmable devices of high SIL (Safety Integrated Level) systems. This explains why the companies which design logical controllers for critical systems are really interested in these methods, and especially in formal verification by model-checking.

In parallel, many researches have been undertaken since several years in order to provide formal timed (Zoubek [2004], Bel Mokadem et al. [2005]) or not timed (Moon [1994], Rausch and Krogh [1998], Jiménez-Fraustro and Rutten [2001], de Smet and Rossi [2002], Huuck [2005]) models of PLC programs, which can be verified with well-known model-checkers, like UPPAAL (Behrmann et al. [2004]) or SMV (McMillan [1999]).

However, formal verification is not at all used during the development of industrial PLC programs up to now (Johnson [2007]). Several reasons can explain this situation: difficulty for automation engineers to write formal properties in temporal logic, absence of automatic translators to formal language in the PLCs development environments, not understandable counterexamples, and, above all, too long, and even infinite, verification time, owing to the classical problem of state space explosion.

The aim of this work is to tackle out (or to limit) state space explosion by proposing a compact formal representation of the behavior of PLC programs. This representation is obtained by using two abstractions: interpretation abstraction, which reduces the number of states, and data abstraction, which lessens the number of the variables which characterize each state. In both cases, the abstract model will keep all the information that are useful for verification. The first results of this research, which were related only to the reduction of the number of states, have been presented in Gourcuff et al. [2006]. This paper presents the overall results and shows the benefits of the two abstractions.

This paper is organized as follows. The next section describes the context of this work: model-checking of industrial PLC programs. The principles which are retained for the construction of the formal model are described in section 3. This allows, in section 4, to expose the method of construction of this compact formal model from a PLC program. The effectiveness of this formal representation is quantified in section 5 on the basis of three examples.

2. CONTEXT

PLCs are automation components which receive information from the process which must be controlled, via their inputs, and send orders via their outputs. Their software is composed of an application program and a scheduler (figure 1). The program is written in one of the languages specified by the IEC 61131-3 standard (IEC [1993]) and is controlled by the scheduler that performs a scanning cycle with three phases: input reading, program treatment and output updating.

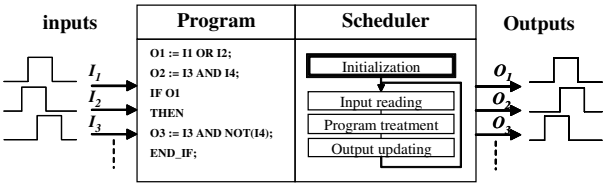

Fig. 1. Structure of a PLC

In the rest of this paper, it will be assumed that the programs respect the following hypotheses:

Hypothesis 1. no use of unlimited loop (while, repeat, for) no jump instruction (goto, break, ...).

⋆ This work was carried out in the frame of a research project which was funded by Alstom Power Plant Information and Control Systems, Engineering tools Department.

Hypothesis 2. only boolean variables are used.

Hypothesis 3. only the textual languages of the IEC 61131-3 standard (ST and IL) are authorized.

Hypothesis 4. multiple assignments of the same variable are possible.

The first hypothesis comes from the good practices of PLCs programming: unlimited loops can indeed lead to too long execution times, which do not fit response time constraints. Only the textual languages are considered in this paper, for room reasons; however, the programs in graphical languages (SFC, FBD and LD) can be easily translated into equivalent ones in textual language (see for instance Machado et al. [2006] for the translation of SFC). The last point, although not advised in certain programming guides, rises from the industrial practice of re-use of parts of codes.

Model-checking of logical controllers is a formal verification that can be integrated in existing PLC program development environment, in opposition to formal synthesis that requires to modify completely the development process. Model-checking can be carried out by many methods presented and classified in Mader [2000], Frey and Litz [2000]. According to those classifications, this paper is limited to non-timed model-checking, without modeling of the controlled process but taking into account the scanning cycle of the controller.

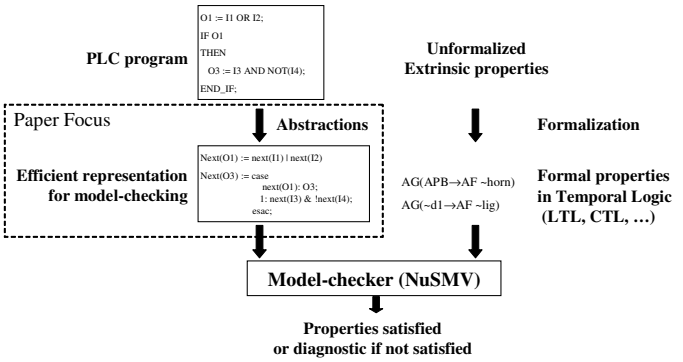

Fig. 2. Objective of the work

This work (illustrated figure 2) is aiming at producing formal models in NuSMV syntax (Cimatti et al. [2002]) which is based on a General Transition System language. Hence, a formal NuSMV model is defined as a transition system which allows to compute the *future* (or next) state from the *current* state. In this paper, the current value (i) of a variable V_j will be noted $V_{j,i}$ and the future value ($i+1$) will be noted $V_{j,i+1}$.

Last, it matters to highlight that focus is put in this paper only on extrinsic properties proof. Two kinds of properties of PLC programs may indeed be verified:

Intrinsic properties , such as absence of deadlock, no blocking state,..., which refer to the behavior of the controller independently of its environment;

Extrinsic properties which refer to the behavior of inputs and outputs, e.g. commission of outputs for a given combination of inputs, always forbidden combination of outputs, expected sequences of inputs-outputs...

As focus is put on extrinsic properties, it becomes possible to construct compact formal models of PLC programs by using the two abstraction techniques which are presented in the next section.

3. ABSTRACTIONS FOR VERIFICATION

3.1 Interpretation abstraction

In the context of formal verification, the overall objective of this abstraction is to reduce the number of states of the formal model which represents the system which must be checked or the number of interactions between states, by keeping only those which are necessary for proving. Several techniques, such as cones of influence (Berezin et al. [1998]), static and dynamic program slicing (Tip [1994]) (Korel and Laski [1988]) have been proposed to reach this objective. For all these approaches, state space reduction is made once the property to prove is given. This implies that as many abstract formal models as properties to prove must be constructed.

The novelty of our approach is to build only one abstract model which will be used for verification of all extrinsic properties. A detailed description of the method which was developed to construct this model from a PLC program will be given in section 4 but an intuitive presentation will be proposed using the example of Figure 3a).

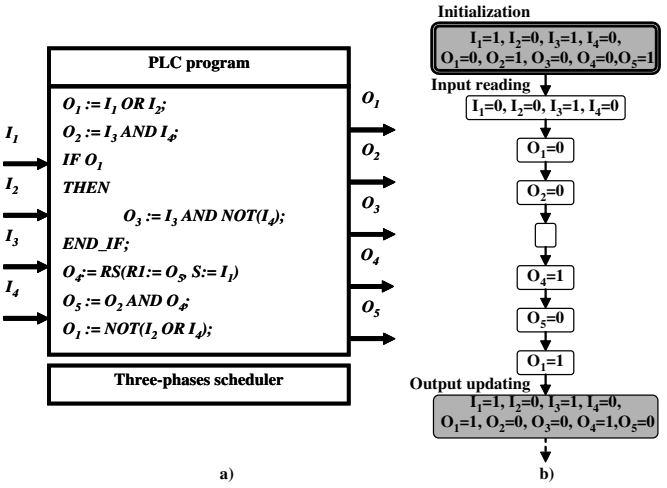

Fig. 3. PLC program example and one possible execution trace

As proposed by de Smet and Rossi [2002], each execution of this ST PLC program can be described as a sequence of states (Figure 3b)), where one state represents the values of the variables after the execution of one program line. In this figure, only the value of the variable that is modified is displayed for each state. It must be noted that the value of the input variables can change only during the input-reading phase.

If only formal verification of extrinsic properties is considered, it becomes obvious that this representation includes two categories of states:

relevant states (grayed on figure 3b)), states which are meaningful for the proof of extrinsic properties (initialization and output updating),

not relevant states, which are the other states.

Checking extrinsic properties requires indeed considering the values of the input/output variables only at the end of the scanning cycle, because this kind of properties focuses on the real values of input and output signals and not on their software representation. The sequentiality of calculation, taken into account in work of Zoubek [2004], de Smet and Rossi [2002], Huuck [2005] can be abstracted in the case of verification of extrinsic properties.

Hence, the principle of the interpretation abstraction that will be adopted in what follows is to represent each scanning cycle by one and only one state which is the relevant state of this cycle. Not relevant (or intermediate) states will not appear in the abstract formal model and one PLC cycle will be treated in one computation step of the model-checker. It can be noted that Moon [1994] mentions the usefulness of this principle for a subset of the Ladder Diagram language (only relays and contacts were considered), but unfortunately does not propose any sound method to apply this principle so as to construct compact representations of PLC programs. This gap will be filled in Section 4.

3.2 Data Abstraction

This abstraction is aiming at decreasing the number of variables which characterize each state of the formal model. Clarke et al. [1994] describes several data abstractions which can be used for reducing the size of formal models to be checked. Several of them focus on abstraction of integer data and then are out of scope of this paper.

For PLC programs with only logic variables, it will be proposed a novel data abstraction which relies on identification of two kinds of variables:

- variables whose current and future values must be kept in the formal model;
- variables whose current value is not necessary.

The variables of the first category will be named R-variables because they are involved in the *R*ecurrence statements that define the transition system. They are formally defined as follows.

Definition 5. A variable V_i of a model M is a *R-variable* if and only if it respects the following property:

$$\exists V_j \in M, V_{j,k+1} = f(\ldots, V_{i,k}, \ldots)$$

where f is a logic function whose one of the arguments is $V_{i,k}$.

The variables of the second category will be named NR-variables because they are *N*ot involved in *R*ecurrence statements that define the transition system.

According to definition 5, we can note that the variables O_1 and O_2 of the example are NR-variables. At each cycle, the values of these two variables are computed from the values of the inputs at the beginning of this cycle (first two lines). The condition in the IF structure refers to the value of O_1 but, as O_1 was previously assigned, only the values of the two inputs I_1 and I_2 for this cycle are necessary to compute this condition, and not the value of O_1 at the previous cycle; then O_1 will not be involved in the formal statement which will define O_3. In a similar way, only the values of the two inputs I_3 and I_4 for this cycle are necessary to compute the value of O_5 (eighth line), and not the value of O_2 at the previous cycle; therefore O_2 will not be involved in the formal statement which will define O_5. To sum up, only the future value of these two variables O_1 and O_2 can be kept in the formal model.

On the other hand, two values of each one of variables O_3, O_4 and O_5 must appear in the formal model because these three variables are R-variables. Analysis of the example and in particular of the conditional structure and the assignment of O_3 shows indeed that:

- $O_{3,k}$ is mandatory to compute $O_{3,k+1}$ (if the condition in the IF structure is false, $O_{3,k}$ is assigned to $O_{3,k+1}$);
- $O_{4,k}$ is mandatory to compute $O_{4,k+1}$ (O_4 is the output of a reset dominant memory (RS) function block and if the two arguments of this block are false, $O_{4,k}$ is assigned to $O_{4,k+1}$);
- $O_{5,k}$ is mandatory to compute $O_{4,k+1}$ (O_5 is an argument of the RS function block and, as the assignment of O_5 stands after the assignment of O_4, $O_{4,k+1}$ is computed from $O_{5,k}$).

This discussion leads to propose the following practical definition, which permits to find out the R-variables when analyzing a PLC program.

Definition 6. A variable is a R-variable if it is used before being assigned, e.g. O_5 in the example, or if it is an argument of its own assignment, e.g. O_3 and O_4.

The two abstractions which have been proposed (relevant states and R-variables) permit an efficient representation of PLC programs that keeps all the necessary information for verification of extrinsic properties. The following section presents the method which was developed, on the basis of these two abstractions, to obtain automatically from a PLC program a formal model which can be checked.

4. FORMAL MODEL CONSTRUCTION

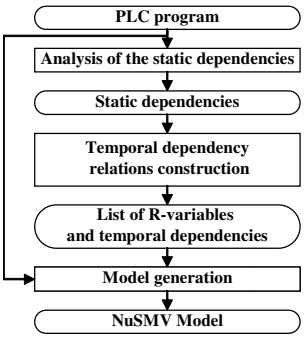

Fig. 4. Translation method

During this research, a software tool was developed to translate automatically PLC programs into compact formal models in the syntax of the model-checker. The algorithms which underlie this translation tool are exposed in this section. It is reminded that the selected model-checker is NuSMV; hence the translation tool will have to yield a set of recurrence equations that models a transition system. Figure 4 shows an overview of the translation

method; its three steps are detailed in the following subsections.

4.1 Analysis of the static dependencies

This analysis is aimed at obtaining, from the PLC program, the static dependency relations between the variables. Each static dependency relation means that a variable is computed from other ones. Those relations are ranked according to the execution order of the program (from top to bottom). Figure 5a) presents the static dependency relations of the example of the figure 3a). On this figure, an arrow from a variable X to a variable Y indicates that Y depends on X, i.e. that X is present in the expression of Y.

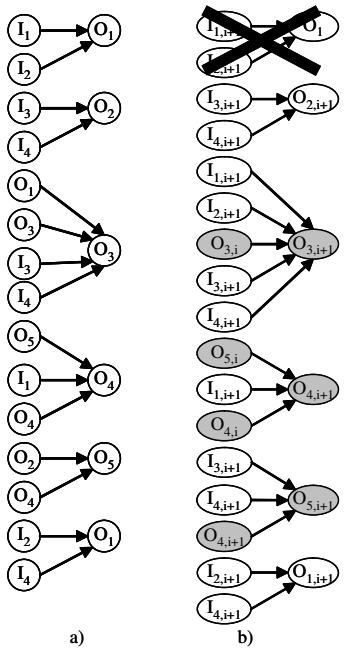

Fig. 5. a) Static and b) temporal dependency relations for the program figure 3a)

Industrial PLC programs often include function blocks (FB), like the RS function block in the example. Each one of the output variables of a given FB may depend not only on the arguments of this FB but also on its internal variables. The static dependency relations of the FB output variables shall account these two types of variables (arguments and internal variables). This implies that detailed models of the FBs which are encompassed in the programs would be developed prior to static dependency relations construction. During this study, all the standard IEC 61131-3 FBs were modeled.

4.2 Temporal dependency relations construction

At the beginning of this step, R-variables and NR-variables are detected using definition 6. Then, an unordered set of temporal dependency relations is obtained from the ordered set of static dependency relations. Each temporal dependency relation is derived from one static dependency relation by replacing the assigned output variable and the input variables by their future values; the value of an output variable at cycle $j + 1$ is computed indeed only from values of input variables at this same cycle. The value of the output variables that are used to define other output variables are determined by algorithm 1. This algorithm replaces each R-variable which an output depends on by either its current value, or its future value, or its later dependency relation according to its previous assignment. NR-variables are replaced by their later dependency relation.

foreach *static dependency relation of the variable* O_k **do**
  foreach *variables* O_j *which* O_k *depends on* **do**
    if O_j *is a R-variable* **then**
      if *its previous assignment* **:**
        case *does not exist*
          Replace O_j by its current value $O_{j,i}$
        case *was the last one*
          Replace O_j by its future value $O_{j,i+1}$
        otherwise
          Replace O_j by its later dependency relation
    else if O_j *is a NR-variable* **then**
      Replace O_j by its later dependency relation

Algorithm 1. Transform_Dependencies(**static dependencies**) → **temporal dependencies**

Last, useless dependency relations, i.e. relations that come from assignments which are superseded by other ones during the program execution (case of multiple assignments of the same variable), are removed from the set of temporal dependency relations.

Figure 5b) shows the result of this step for the example figure 3a). It can be noted that the first dependency relation of O_1 allows constructing the temporal dependency relation of O_3, but is not kept at the end; only the last dependency relation of O_1 is preserved. This figure shows also that O_3, O_4, O_5, R-variables (grayed on figure 5b)) according to definition 6, respect definition 5: their current values are used to compute future values of variables.

4.3 Model generation

Once the final set of temporal dependency relations obtained, the NuSMV model that represents, in an abstract fashion, the PLC program is constructed as sketched in what follows.

- Each PLC program statement (assignment or conditional structure) that defines a R-variable gives rise to one NuSMV assignment. This later assignment describes formally the behavior of the statement and includes variables values that are given by the temporal dependency relation that corresponds to the statement.
- The statements that define NR-variables are not translated into NuSMV assignments. They are put aside as expressions to be used when a property to prove contains these variables. Hence NR-variables will be replaced by their expressions in properties.
- Function-blocks calls are replaced by instances of formal generic models of the corresponding FBs. These generic models are stored in a FBs library which was filled in beforehand. For instance, the generic form of the *RS* function block, with two input

variables S and $R1$ and one output variable Q, is: $Next(Q) := !R1 \ \& \ (S \mid Q)$;.

Figure 6 presents the result of the translation method for the example. Due to limited place, only the core of the model, the output definition, is presented in this figure. Variables declarations and initialization are not given. It matters to underline the reduced size (3 statements) of the NuSMV formal model.

NuSMV model of the PLC program:

$Next(O_3) := case$
$Next(I_1) \mid Next(I_2) \ : \ Next(I_3) \ \& \ !(Next(I_4));$
$!(Next(I_1) \mid Next(I_2)) \ : \ O_3;$
$esac;$
$Next(O_4) := !Next(I_1) \ \& \ (O_5 \mid O_4)$
$Next(O_5) := Next(I_3) \ \& \ Next(I_4) \ \& \ Next(O_4);$

Expressions to be used for properties:

$O_2 := Next(I_3) \ \& \ Next(I_4);$
$O_1 := !(Next(I_2) \mid Next(I_4));$

Fig. 6. Formal model of the example

5. REPRESENTATION EFFECTIVENESS EVALUATION

This section presents the results of three case studies whose overall objective is to quantify the effectiveness of the proposed representation and evaluate its benefits for formal verification of large-sized PLC programs. The first two case studies are aiming at comparing the representation which is described in this paper with those proposed in de Smet and Rossi [2002] (without abstraction) and Gourcuff et al. [2006] (with only interpretation abstraction), according to several criteria (state space size, time required to prove properties, ...). The last case study focuses on the applicability of this representation to real industrial programs.

5.1 First case study

This study is based on the simple example of figure 3a). From the example, three formal models were automatically constructed by means of a software tool which used either no abstraction, or only the interpretation abstraction or the two abstractions. A behavior-equivalence analysis was performed between these three models and was positive: whatever the inputs sequence, the three models emit the same outputs sequence.

Representation given in	Reachable states	Modeling variables	System diameter
de Smet and Rossi [2002]	314 out of 14336	7	22
Gourcuff et al. [2006]	21 out of 512	5	3
This paper	20 out of 128	3	3

Table 1. Comparison of three representations for the example of figure 3a)

The following features of the three models are displayed in table 1:

Reachable states: I out of J means that I states are really reachable among the J possible ones (combination of all the possible states of the variables).

Modeling variables: number of variables (except inputs) that the model-checker must memorize at each evolution. As seen in section 4, these variables are the R-variables when using data abstraction.

System diameter: number of evolutions necessary to explore the whole reachable state space. This feature impacts the maximum size of counterexamples.

These results show clearly that interpretation abstraction reduces significantly the size of the state space, that is roughly divided by 15, and the system diameter. This abstraction allows also slightly lessening the number of modeling variables because the variables which modeled the execution of the program in de Smet and Rossi [2002] are no more helpful.

As it could be easily forecast, data abstraction reduces the number of modeling variables; only the three R-variables are necessary, as explained above. Reducing the number of modeling variables leads to a (weak for this toy example) reduction of the number of reachable states. The number of initial states decreases indeed when the number of modeling variables is reduced.

Last, it can be pinpointed that the two abstractions will allow shorter counterexamples (the maximum size of a counterexample is related to the system diameter) that will ease analysis in case of negative proof. This is an indirect, but positive, consequence of states and modeling variables numbers reduction.

5.2 Second case study

The objective of this second case study is to compare the time and memory performances of the three representations when checking safety and liveness properties. This experimentation is based on the system presented and checked in de Smet and Rossi [2002]: a controller of Fischertechnik system. The same proofs were made by using the model-checker NuSMV, version 2.3.1, on a PC P4 3.2 GHz, with 1 GB of RAM, under Windows XP.

Representation given in	liveness properties	safety properties
de Smet and Rossi [2002]	5h / 526MB	20min / 200MB
Gourcuff et al. [2006]	2s / 8MB	2s / 8MB
This paper	0.3s / 8MB	0.3s / 8MB

Table 2. Time and memory required for properties verification

The benefits of the two abstractions are significant. Interpretation abstraction alone allows reducing the duration of a proof by a factor 4000 to 60000 and the necessary memory size by a factor 40 to 60, compared to the representation of de Smet and Rossi [2002]. As the minimum memory size to run NuSMV in the conditions of this experiment (8 MB) is already reached with only the interpretation abstraction, the benefit of data abstraction for memory size reduction cannot be shown; however this later abstraction still improves the time performances by one order of magnitude.

5.3 Third case study

This third case study focuses on the real control system of a power plant which includes 175 networked PLCs. The

main objective of this experimentation was to determine, for each one of these PLCs, whether the model-checker was able to compute the size of the reachable state space of the formal model obtained by using the translation method of section 4. If it is possible to compute the whole reachable state space, then it is also possible to check most of the properties on the formal model without encountering combinatory explosion. Table 3 synthesizes the results.

Number of inputs	max: 50 min: 2 sum: 2317
Number of outputs	max: 47 min: 1 sum: 1822
Number of R-variables	max: 18 min: 1 sum: 238
Size of the reachable state space	max: 8.10^{28} min: 10^5 mean: 5.10^{26}
Whole state space exploration time	1 sec for 175 programs

Table 3. Statistics obtained for a set of 175 industrial PLC programs

The first two lines define the industrial constraints by giving the maximum, minimum and overall number of input and output variables of the programs. The third line shows that data abstraction divides the number of modeling variables of the formal model roughly by 8 (only 238 R-variables are necessary compared to the 1822 output variables that would require a formal model which would not use this abstraction). The maximum, minimum and mean size of the state spaces are given at the fourth line; it matters to underline that the model-checker was able to explore all the formal models obtained from the 175 PLCs, that is not possible when no abstraction is used. Last, the time to explore the whole set of these state spaces is displayed at the fifth line; this small value leads to plan short verification time for these industrial programs.

Those three case studies show clearly the effectiveness of the representation based on the two abstractions, compared to previous ones, and its usefulness when dealing with real cases.

6. CONCLUSION

Formal verification of PLC programs can strongly contribute to improve safety and dependability of automated systems. However, transferring these formal methods to the industrial world requires methods that deliver formal models whose size is small enough to avoid combinatory explosion and permit verification in reasonable times, when large-sized programs are dealt with.

This paper proposes such a method for verification of extrinsic properties, a major concern in industry. This method is based on interpretation and data abstractions and provides models which contain all the useful information for verification. The efficiency of the proposed representation of PLC programs, compared to previous ones, has been assessed and the ability to translate industrial programs into compact formal models has been shown.

On-going works address formal verification of industrial controllers that do not only perform logic control tasks. This will lead to investigate abstraction techniques for hybrid systems.

REFERENCES

IEC Standard 61131-3 : Programmable controllers - Part 3, 1993.

IEC Standard 61508 : Functional safety of electrical/electronic/programmable electronic safety-related systems, 2000.

G. Behrmann, A. David, and K.G. Larsen. A tutorial on UPPAAL. *LNCS*, 3185:200–236, September 2004.

H. Bel Mokadem, B. Bérard, V. Gourcuff, J.-M. Roussel, and O. de Smet. Verification of a timed multitask system with Uppaal. In *ETFA'05*, pages 347–354, Catania, Italy, September 2005.

S. Berezin, S. Campos, and E. M. Clarke. Compositional reasoning in model checking. *LNCS*, 1536:81–102, 1998.

A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. CAV 2002*, volume 2004 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

O. de Smet and O. Rossi. Verification of a controller for a flexible manufacturing line written in ladder diagram via model-checking. In *ACC'02*, pages 4147–4152, Anchorage (USA), May 2002.

G. Frey and L. Litz. Formal methods in PLC programming. In *Proceedings of the IEEE SMC 2000*, pages 2431–2436, October 2000.

V. Gourcuff, O. de Smet, and J-M. Faure. Efficient representation for formal verification of PLC programs. In *WODES'06*, pages 182–187, July 2006.

R. Huuck. Semantics and Analysis of Instruction List Programs. In *SFEDL'2004*, pages 3–18, January 2005.

F. Jiménez-Fraustro and É. Rutten. A Synchronous Model of IEC 61131 PLC Languages in SIGNAL. In *ECRTS*, pages 135–142, 2001.

T.L. Johnson. Improving automation software dependability: A role for formal methods? *Control Engineering Practice*, 15(11):1403–1415, 2007.

B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

J. Machado, B. Denis, J.-J. Lesage, J.-M. Faure, and J. C. L. Ferreira Da Silva. Logic controllers dependability verification using a plant model. In *DESDes'06*, pages 37–42, Rydzyna (Poland), September 2006.

A. Mader. A classification of PLC models and applications. In *WODES'2000*, pages 239–247, August 21-23 2000.

K. L. McMillan. *The SMV Language*. Cadence Berkeley Labs, 1999.

I. Moon. Modeling programmable logic controllers for logic verification. In *Control Systems Magazine, IEEE*, pages 53–59. IEEE Comp. Soc. Press, 1994.

M. Rausch and B. Krogh. Formal verification of PLC programs. In *American Control Conference*, pages 234–238, PA, USA, June 1998.

F. Tip. *A Survey of Program Slicing Techniques*. Centrum voor Wiskunde en Informatica, 1994.

B. Zoubek. *Automatic verification of temporal and timed properties of control programs*. PhD thesis, University of Birmingham, 2004.