Towards the automatic verification of PLC programs written in Instruction List

G. Canet[1,3], S. Couffin[2], J.-J. Lesage[2], A. Petit[3] and Ph. Schnoebelen[3]

(1)Alcatel CIT
Corporate Research Center
F-91461 Marcoussis Cedex

(2)LURPA
ENS de Cachan
F-94235 Cachan Cedex

(3)LSV
UMR 8643 CNRS & ENS de Cachan
F-94235 Cachan Cedex

Abstract—**We propose a framework for the automatic verification of PLC (Programmable Logic Controllers) programs written in Instruction List, one of the five languages defined in the IEC 61131-3 standard. We propose a formal semantics for a significant fragment of the IL language, and a direct coding of this semantics into a model checking tool. We then automatically verify rich behavioral properties written in linear temporal logic. Our approach is illustrated on the example of the tool-holder of a turning center.**

Keywords— **Verification, Instruction List, IEC 61131-3, model checking, operational semantics.**

I. Introduction

With the emergence of standardized programming languages for PLCs (the IEC 61131-3 standard [6]), the interest in general verification methods based on formal models [9] is growing. In this paper we consider programs written in IL (Instruction List) and describe our approach to the exhaustive formal verification of complex control applications. In this first work, we only focus on so-called *simple* programs, i.e. programs made of one module, only handling "Boolean" or "bounded integer" variables, and no timer. As exemplified below, such simple IL programs often occur as part of a larger program.

Our approach combines two formal components: an operational semantics for IL programs, and a temporal logic in which we state properties to be checked. The verification itself is performed by the Cadence SMV[1] [11] model checker. In order to verify a system by means of model checking, we first build a formal model of it and we formally state the expected behavioral properties, using a language for property specification, a temporal logic for instance. Then a model checking algorithm is able to say whether the (model of the) system satisfies the properties or not (Fig. 1). When a property is not satisfied, most model checkers give an accompanying diagnosis, e.g. an example of a system execution violating a property. A symbolic model checker like SMV does not represent explicitly the whole transition system, but uses efficient symbolic representation techniques [12].

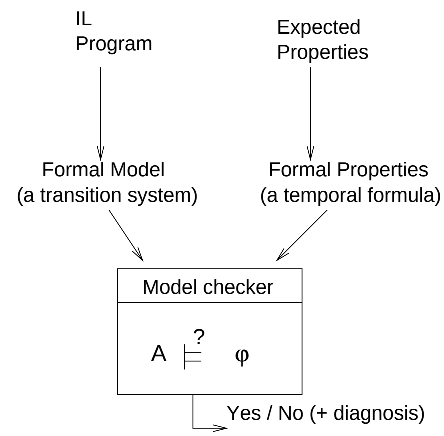

Fig. 1. General model checking scheme.

The operational semantics of IL programs is given under the form of a transition system. A *configuration* (or a *state*) of the program is formally represented by a tuple containing the assignment of current values to all variables, including implicit variables. Transitions between these states formalize the effect of executing one instruction (or acquiring the input variables). The resulting transition system is described in Cadence SMV input language via a simple translation/compilation tool running on the original IL program.

The expected properties are then written as LTL (Linear Temporal Logic, [2]) formulas, the temporal language handled by Cadence SMV.

We illustrate our approach with the complete treatment of the example of the control of the tool-holder of a turning center. Some expected properties of the system are verified. These properties can be invariants, safety properties, liveness properties, possibly nested and combined in arbitrary way. We will for instance show how behavioral properties can be verified.

The breakthrough aspect of our work relies on the fact that it combines (1) a completely formal, but nevertheless

[1] We chose this model checker because of its powerful symbolic techniques, but our approach could be adapted to any model checker aware of transitions systems.

relatively simple, semantics of a significative fragment of the IL language, (2) a direct coding of this semantics into a model-checking tool, allowing (3) the automatic verification of behavioral properties (more complex than reachability, or invariant-based properties). If some models of IL programs, based on timed automata [10], Petri Nets ([5] and [4]), Higher Order Logic [15], synchronous languages [7] or Condition/Event systems ([3] and [8]) have been proposed, our approach makes it possible to deal fully automatically and in-depth with some real examples of a non elementary size.

II. Definition and operational semantics of the IL language

A. *Definition of a program written in IL*

An IL program consists of a declaration of variables followed by a program body containing instructions.

A.1 Declaration of variables

Declaring a variable means giving a symbol $\mathtt{x}$ for the variable, its type $\mathrm{t}(\mathtt{x})$ and an optional initial value $\mathrm{init}(\mathtt{x})$. In this paper we only consider booleans and bounded integers. A type t of bounded integer is defined by two integers min_t and max_t, with $\mathtt{minint} \leq \mathtt{min_t} < \mathtt{max_t} \leq \mathtt{maxint}$, where `minint` and `maxint` are constants (depending on the hardware, e.g. -2^{15} and $2^{15}-1$).

A.2 Program body

The body of an IL program is a finite sequence of command lines. Each command line is a couple (l_i, ins_i), where l_i is an optional label and ins_i is an instruction. An instruction is an operator followed by an optional argument.

IL operators perform some computations over the variables of the program, using mostly the accumulator, (also called the "current result"). The main instructions are:

`LD x`	loads the accumulator with the value of `x` (`LDN` loads $\neg\mathtt{x}$);
`ST x`	stores the value of the accumulator into `x`;
`AND x`	computes a logical 'and' between `x` and the accumulator, then stores the result into the accumulator;
`EQ x`	checks if the value of the accumulator is equal to `x`, and loads the result (a Boolean) into the accumulator;
`LT x`	checks if the accumulator is strictly lower than `x`, and loads the result into the accumulator;
`JMP l`	jumps to label `l` (`JMPC` : jumps only if the value of the accumulator is $True$, `JMPCN` only if it is $False$);
`RET`	terminates the program execution (`RETC` terminates only if the value of the accumulator is $True$, `RETCN` if it is $False$).

Every declaration followed by a sequence of command lines does not give necessarily a well formed program. There exist several simple rules (unicity of declaration, type consistency,...) by which one can decide whether a given program is well formed or not. An IL compiler requires these rules to be fulfilled.

B. *Operational semantics*

We consider an IL program composed of k lines. Let Σ be the set of all the variable names declared in the program, and E be the subset of Σ containing the names of the input variables. A valuation of Σ is a function which associates a value with any variable $\mathtt{x}$ in Σ. A value is $False$ or $True$ if $\mathtt{x}$ is a Boolean, or an element of $[min_{\mathrm{t}(\mathtt{x})}, max_{\mathrm{t}(\mathtt{x})}]$ if $\mathtt{x}$ is a bounded integer.

B.1 Cyclic behavior

The industrial controllers that we are interested in behave in a cyclic way. The three steps of the cycle are:

1. input acquisition;
2. program execution;
3. output assignment.

In our formal model, we define an *end of cycle* phase, which indicates that the program execution has terminated. This is the phase in which the outputs can be observed. This phase precedes a new input acquisition, and so on.

B.2 Transition system associated with an IL program

We represent the behavior of an IL program by a transition system $\langle Q, \rightarrow, I\rangle$, where Q is the set of states (also called *configurations*). Q contains tuples of the form (V, a, m), where:

- V is a valuation of Σ;
- $a \in \{False, True\} \cup [\mathtt{minint}, \mathtt{maxint}]$ is the value of the accumulator;
- $m \in [1, k+1] \cup \{err\}$ is the value of the program counter.

$I \subseteq Q$ is the set of the *initial states*: (V, a, m) is initial if and only if the program counter m equals 1, and if $V(\mathtt{x}) = \mathrm{init}(\mathtt{x})$ for all variable $\mathtt{x}$ for which an initial value was declared.

A state (V, a, m) with $1 \leq m \leq k$ corresponds to the state of the system before executing the line m of the IL program. The states of the form $(V, a, k+1)$ are the end-of-cycle states.

There remains to define the transition relation $\rightarrow \subseteq Q \times Q$ accounting for the IL program. Each program line gives rise to a set of transitions: line i yields transitions starting from states of the form (V, a, i), (i.e. where the program counter equals i) and going to some (V', a', i'). Usually

Fig. 2. A turning center and its tool-holder turret.

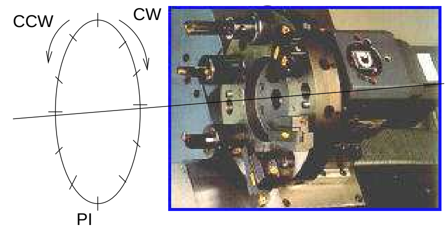

Fig. 3. Detail of the tool-holder turret.

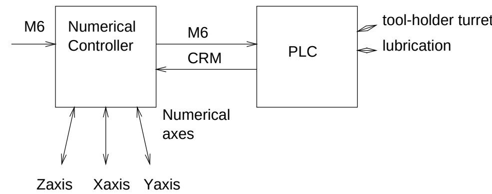

Fig. 4. Global architecture of the turning center control.

V', a', and i' are obtained by modifying V, a and i according to the program line. If, e.g., the line is "`ST s`", then a' is a and V' is like V except that $V'(\mathtt{s})$ now equals a. If the line is "`AND s`" then V' is V unmodified and a' is the conjunction of a and $V(\mathtt{s})$. i' equals $i+1$ unless the line is a `JMP`, `RET`, or some other jump instruction.

In the case of operations over bounded integers, some transitions lead to an error state, e.g. when a value should be assigned to some variable and it overflows the range of the variable. For instance, if the ith command line is "`ST x`" where `x` is a bounded integer variable, there are transitions $(V, a, i) \rightarrow (V, a, err)$ for all $a \notin [min_{\mathtt{t(x)}}, max_{\mathtt{t(x)}}]$.

Finally there are transitions representing the acquisition of the input variables (we assume that during this phase input variables can take arbitrary values): for all valuations V and V', if V and V' only differ on input variables (formally, $\mathtt{x} \in \Sigma \setminus \mathtt{E} \Rightarrow \mathtt{V'(x)} = \mathtt{V(x)}$) then there exists a transition $(V, a, k+1) \rightarrow (V', a, 1)$.

We do not explicitly represent the output assignment phase, but the output variables can be observed in all states (V, a, m) where $m = k+1$.

III. Example

A. Description

The example deals with the control of the tool-holder turret (fig. 3) of a turning center (as represented in fig. 2). The tool-holder turret is for twelve places (for live or fixed tools), it can rotate clockwise (`CW`) or anti-clockwise (`CCW`). The control has to minimize the time for tool changing.

The global architecture of the turning center, represented in figure 4, is made of two major components:

- the numerical controller manages the interaction with the user, and performs all the real-time computations that require some precision (like managing the axes);
- the PLC controls the operations that require more complex and flexible computations. In our case, the management of the position of the tool-holder turret is made by a program implemented in the PLC.

When the numerical control director of the turning center receives the code `M6` (the tool changing order), it transmits it to the PLC, which computes at each PLC cycle the current position of the turret (integer position from a 4-bit coding position), determines the rotating sense (`RH` or `RAH`), defines whether the indexing position (the position just following the goal position) is reached (`PI`), and controls the turret actuators. When the tool is changed, the order `CRM` is sent. Sensors give information about the current position of the turret (4-bit coding position), the indexing of the turret (`CCI`) and the locking of the turret (`CCB`).

B. Tool changing program

The tool changing program (in SFC) is represented in figure 5.

We focus on the Action4 of the program. This action is described in IL. The aim of this program is to control the turret actuators (rotating motor `CW` and `CCW`, indexing electro-magnet `EAI`, brake `Br`) using the rotating sense (`RH` and `RAH` calculated previously by Action2), depending on the results of Action1 and Action3 (these actions are performed in parallel with Action4 to determine the indexing position and if it is reached) and with regards to the value of the sensors (current position reached `PI`, indexing `CCI` and locking of the turret `CCB`). When the tool has been changed, the order `CRM` is sent.

The body of IL program, without the variable declarations, can be found below. In the following code we add

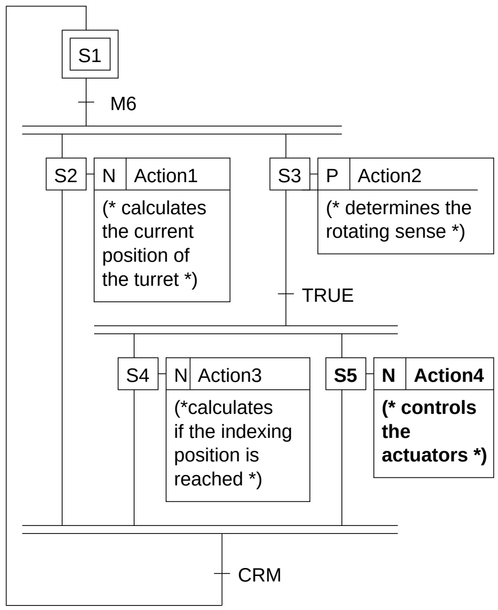

Fig. 5. Tool changing program

line numbers for readability; they are not part of the program.

```
l1:              LD       x1
                 JMPCN    l2
                 LD       FALSE
                 ST       CRM
                 ST       Br
                 LD       RH
                 JMPCN    l1_test_rah
                 LD       FALSE
                 ST       CCW
                 LD       TRUE
                 ST       CW
l1_test_rah:     LD       RAH
                 JMPCN    l1_trans
                 LD       TRUE
                 ST       CCW
                 LD       FALSE
            ST   CW
l1_trans:        LD       PI
                 RETCN
                 LD       TRUE
                 ST       x2
                 LD       FALSE
                 ST       x1
                 RET
l2:              LD       x2
                 JMPCN    l3
                 LD       strobe
                 RETC
                 LD       TRUE
                 ST       x3
                 ST       EAI
                 LD       FALSE
                 ST       x2
                 RET
l3:              LD       x3
                 JMPCN    l4
                 LD       CCI
                 RETCN
                 LD       TRUE
                 ST       x4
                 LD       FALSE
                 ST       x3
                 RET
l4:              LD       x4
                 JMPCN    l5
                 LD       t1
                 RETCN
                 LD       RH
                 JMPCN    l4_test_rah
                 LD       TRUE
                 ST       CCW
                 LD       FALSE
                 ST       CW
l4_test_rah:     LD       RAH
                 JMPCN    l4_trans
                 LD       FALSE
                 ST       CCW
                 LD       TRUE
                 ST       CW
l4_trans:        LD       TRUE
                 ST       x5
                 LD       FALSE
                 ST       x4
                 RET
l5:              LD       x5
                 JMPCN    l6
                 LD       CCB
                 RETCN
                 LD       TRUE
                 ST       x6
                 ST       Br
                 LD       FALSE
                 ST       x5
                 ST       CW
                 ST       CCW
                 RET
l6:              LD       x6
      RETCN
                 LD       t2
                 JMPCN    l6_trans
                 LD       FALSE
                 ST       EAI
l6_trans:        LD       CCI
                 RETC
                 LD       TRUE
                 ST       CRM
                 LD       FALSE
                 ST       x6
                 ST       EAI
```

IV. Program verification by means of model checking

A. Coding the operational semantics

We use the model checking technique ([1], [12], [14]) to verify behavioral properties of the system. In this work, we used Cadence SMV [11]. The main advantage of SMV is the fact that it relies on powerful symbolic representation techniques. SMV is therefore able to verify systems on large scale.

In our coding, we declare additional variables representing the accumulator and the program counter. For instance, the `pc` variable, corresponding to the current line, is a bounded integer variable, whose values range from 0 to 90 (in our example, the program has 89 actual instruction lines). The value 0 is used to represent the error state.

For instance, the evolution of the `CRM` variable is described as follows in SMV syntax:

```
next(CRM) := switch (pc)
{
```

```
        4          : a;
        86         : a;
        default    : CRM;
    };
```

This can be interpreted as follows: for each transition $(V, a, i) \rightarrow (V', a', i')$ of the operational semantics, we have:

- $V'(\mathtt{CRM}) = \mathtt{a}$ if $i = 4$ or if $i = 86$,
- $V'(\mathtt{CRM}) = \mathtt{V}(\mathtt{CRM})$ otherwise.

The value 4 and 86 correspond to the beginning of the transitions modelling the commands `ST CRM`.

The input variables only change in the transitions coding variable acquisition, when $pc = 90$. For instance, the evolution of `PI` is defined as follows:

```
    next(PI) := switch (pc)
    {
        90         : {0, 1};
        default    : PI;
    };
```

In the case where $pc = 90$, `PI` can take any possible Boolean value.

B. Linear temporal logic

The properties we are interested in are mainly (but not restricted to) the following ones:

- static properties or invariants: "p always holds";
- safety properties: "p holds as long as q holds";
- liveness properties: "p will hold eventually".

We use the LTL logic to verify our properties. LTL is a logic which allows us to write behavioral properties of the system. We call *path* a sequence of states $\pi = s_0, s_1, s_2, \ldots$ where, for all i, there exists a transition $s_i \rightarrow s_{i+1}$. In this study we only consider infinite paths (in fact, in our semantics, every state has at least one successor). For each path, a given LTL property does or does not hold. LTL expressions contain "atomic propositions" (in our case, these are the Boolean variables, or predicates over the integer variables, e.g. $pc = 10$) and Boolean operators. For instance, if a and b are atomic propositions, we say that $a \wedge b$ holds for some path π if a and b are true in the first state π_0 of π.

LTL formulas also contain temporal operators, defined as follows. If φ_1, φ_2 are LTL formulas, $\pi = s_0, s_1, \ldots$ is a path, and π^i the suffix of π starting from the i^{th} state, i.e. $s_i, s_{i+1}, \ldots$, we say that:

- $\mathsf{X}\,\varphi$ holds in π if φ holds in π^1 ("φ holds in the next state");
- $\varphi_1 \,\mathsf{U}\, \varphi_2$ holds in π if φ_2 holds for some π^i and, for all $j < i$, φ_1 holds in π^j ("φ_1 holds until φ_2 eventually holds");
- $\mathsf{F}\,\varphi$ holds in π if φ holds for some π^i;
- $\mathsf{G}\,\varphi$ holds in π if φ holds for all π^i.

C. Properties verification

In this section we show how to formally write some typical expected properties of the above IL program. For each property, SMV automatically checks that it holds along all execution paths. If the property is not satisfied, SMV gives a path as a counterexample.

C.1 Invariant : motor command consistency

We want to verify that the engine is never turned on in both directions at the same time. In other words, there is no state of the program execution where both variables `CW` and `CCW` equal $True$. This property is an invariant that can be written

$$\mathsf{G}\,\neg(\mathtt{CW} \wedge \mathtt{CCW})$$

SMV answers that the property is not verified by our IL program. The inspection of the counter-example run shows why: the problem comes from the fact that the variables `CW` and `CCW` do not strictly represent the values of the output lines. We actually know that the output variables are updated in a separate phase, which takes place after the execution of the program. While the program is running, it is possible that `CW` and `CCW` are true at the same time, but what the program designer really wanted to avoid is `CW` and `CCW` both true at the end of the program execution (when the ouputs are assigned).

Our property was too strictly formalized. Let **eoc** be a proposition representing the end of the execution cycle (in our case **eoc** stands for $(\mathtt{pc} = 90)$). We can write the property we wanted with the formula

$$\mathsf{G}\,(\mathbf{eoc} \rightarrow \neg(\mathtt{CW} \wedge \mathtt{CCW}))$$

SMV answers that this property is satisfied.

C.2 Safety: brake-motor consistency

We want to verify that the motor is always turned off before the brake is on. This is a safety property (*the brake is off as long as the motor is on*). That can be stated by the following formula:

$$\mathsf{G}\,(\neg\mathtt{Br}\,\mathsf{W}\,(\neg\mathtt{CW} \wedge \neg\mathtt{CCW}))$$

The W (or *weak until*) operator is an additional LTL operator. For a given path, the formula $\varphi\,\mathsf{W}\,\psi$ holds if and only if φ continuously holds as long as ψ does not hold. SMV does not allow direct use of W, but we can rewrite it using the equivalence

$$\varphi\,\mathsf{W}\,\psi \equiv (\varphi\,\mathsf{U}\,\psi) \vee \mathsf{G}\,\psi$$

Adding the "end of cycle" synchronization, we submit the following property to SMV:

$$\mathsf{G}\,((\mathbf{eoc} \rightarrow \neg\mathtt{Br})\,\mathsf{U}\,(\neg\mathtt{CW} \wedge \neg\mathtt{CCW} \wedge \mathbf{eoc})) \vee \mathsf{G}\,(\mathbf{eoc} \rightarrow \neg\mathtt{Br})$$

SMV answers that the property is verified.

C.3 Liveness : non-blocking system

We now want to make sure that, if one of the variables RH and RAH (indicating that the motor must be turned on) is set when the program is launched, then the program is properly executed until it sends the ending signal (the CRM variable). We can verify this property by means of the following formula:

$$\mathsf{G}\,((\mathtt{RH} \vee \mathtt{RAH}) \rightarrow \mathsf{F}\,\mathtt{CRM})$$

In fact, this formula is proven false by SMV, because the system is not sufficiently specified. For instance, the system "thinks" that the motor can be infinitely turned on without reaching its goal, which is physically impossible. For the property to hold, we should at least suppose that the variable PI is set at some time. On the opposite, if we suppose that PI is never set, there is no way for the program to come to an end. In fact, PI is an input variable whose evolution depends on physical events. In order to prove the property, we have to make assumptions on the behavior of the input variables, thus supposing the corresponding devices work as expected. There mainly are two kinds of assumptions that we can make:

- "some event x is always true from time to time (it is never infinitely false)". This is called a fairness property. The corresponding LTL formula is $\mathsf{G}\,\mathsf{F}\,\mathsf{x}$ (from every state of the execution, x will eventually hold);
- "it is always true that, if some event x holds, then some other event y will eventually hold". The corresponding formula is $\mathsf{G}\,(\mathsf{x} \rightarrow \mathsf{F}\,\mathsf{y})$.

In order to prove our liveness property, we have to make fairness assumptions on CCI, t1, CCB and $\neg$strobe. We also make the assumption $\mathsf{G}\,((\mathtt{CW} \vee \mathtt{CCW}) \rightarrow \mathsf{F}\,\mathtt{PI})$. This is a physical behavioral property of the motor device: if the motor is on, the device will eventually reach its goal position. After making these assumptions, our liveness property is proven by SMV.

V. Conclusion

We have developed a formal method to perform the verification of PLC programs written in the IL language. This method consists in applying symbolic model checking techniques in the framework of PLC programs. The characteristic elements of our approach are:

- the choice of a significative fragment of the IL language, allowing to write some simple programs;
- a sharp transition system-based operational semantics of this fragment;
- a coding of these transition systems into the input language of a model checker (like Cadence SMV);
- the use of the LTL linear temporal logic to write behavioral properties.

Although based on simple and well-known concepts, this approach allows to prove or reject, in a completely automated way, the correctness of IL programs of a non-trivial size.

A similar study on the validation of PLC programs has already been presented for LD programs in [13]. Our goal is to propose to PLC programs designers a set of simple and efficient methods allowing them to validate their programs. We aim to extend this method to a larger IL fragment, and to programs written in other languages of the IEC 61131-3 standard, especially multilanguage programs.

References

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

[2] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B*, chapter 16, pages 995–1072. Elsevier Science Publishers, 1990.

[3] H.-M. Hanisch, J. Thieme, A. Lüder, and O. Wienhold. Modeling of PLC behavior by means of Timed Net Condition/Event Systems. In *Proc. 6th IEEE Conf. Emerging Technologies and Factory Automation (ETFA'97), Los Angeles*, pages 110–122, 1997.

[4] M. Heiner and T. Menzel. Instruction List verification using a Petri Net semantics. In *IEEE Int. Conf. on Systems, Man and Cybernetics, San Diego, CA, USA, Oct. 1998*, pages 716–721, 1998.

[5] M. Heiner and T. Menzel. A Petri net semantics for the PLC language Instruction List. In *Proc. 4th IEEE Workshop on Discrete Event Systems (WODES'98), Cagliari, Italy, Aug. 1998*, pages 161–165, 1998.

[6] IEC (International Electrotechnical Commission). *IEC Standard 61131-3 : Programmable controllers - Part 3*, 1993.

[7] F. Jiménez-Fraustro and É. Rutten. Modélisation synchrone de standards de programmation de systèmes de contrôle : le langage ST de la norme CEI 1131-3. In *Journée d'études « Nouvelles percées dans les langages pour l'automatique », SEE – Club 18 (Automatique et Automatisation Industrielle), Amiens (France)*, 1999.

[8] S. Kowalewski, S. Engell, and O. Stursberg. Verification of logic controllers for continuous plants. In *Advances in Control : Highlights of ECC'99*, pages 345–389. Springer-Verlag, 1999.

[9] S. Lampérière-Couffin, O. Rossi, J.-M. Roussel, and J.-J. Lesage. Formal validation of PLC programs: a survey. In *European Control Conference 1999 (ECC'99), Karlsruhe, Germany, Aug.-Sep. 1999*, 1999. proceedings on CD-ROM, communication 741.

[10] A. Mader and H. Wupper. Timed automaton models for simple programmable logic controllers. In *Proc. 11th Euromicro Conference on Real-Time Systems (ECRTS'99), York, UK, June 1999*, pages 114–122. IEEE Comp. Soc. Press, 1999.

[11] K. L. McMillan. *The SMV Language*. Cadence Berkeley Labs. http://www-cad.eecs.berkeley.edu/~kenmcmil/language.ps.

[12] K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, 1993.

[13] O. Rossi, O. de Smet, S. Lampérière-Couffin, J.-J. Lesage, H. Papini, and H. Guennec. Formal verification: a tool to improve the safety of control systems. In *4th Symposium on Fault Detection, Supervision and Safety for Technical Processes (IFAC Safeprocess 2000), Budapest, Hungary*, 2000. to appear.

[14] Ph. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, A. Petit, et al. *Vérification de logiciels : Techniques et outils du model-checking*. Vuibert, 1999.

[15] N. Völker and B.J. Krämer. Modular verification of function block based industrial control systems. In *Proc. 24th IFAC/IFIP Workshop on Real-Time Programming (WRTP'99), Dagstuhl, Germany, May-June 1999*. IFAC, 1999.