EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN-ACC-NOTE-2014-0037
10 June 2014

Automated Generation of Formal Models from ST Control Programs for Verification Purposes

Borja Fernández Adiego[1,4)], Dániel Darvas[1,2)], Jean-Charles Tournier[1)],
Enrique Blanco Viñuela[1)], Jan Olaf Blech[3)], Víctor M. González Suárez[4)]

Abstract

In large industrial control systems such as the ones installed at CERN, one of the main issues is the ability to verify the correct behaviour of the Programmable Logic Controller (PLC) programs. While manual and automated testing can achieve good results, some obvious problems remain unsolved such as the difficulty to check safety or liveness properties. This paper proposes a general methodology and a tool to verify PLC programs by automatically generating formal models for different model checkers out of ST code. The proposed methodology defines an automata-based formalism used as intermediate model (IM) to transform PLC programs written in ST language into different formal models for verification purposes. A tool based on Xtext has been implemented that automatically generates models for the NuSMV and UPPAAL model checkers and the BIP framework.

Keywords: PLC, ST, automata, verification, model checking, NuSMV

CERN Internal Note

[1]CERN (EN/ICE), Geneva, Switzerland. E-mails: {ddarvas, bfernand, jtournie, eblanco}@cern.ch.
[2]Budapest University of Technology and Economics, Budapest, Hungary.
[3]RMIT University, Melbourne, Australia.
[4]University of Oviedo, Gijón, Spain.

Contents

1 Introduction

CERN, the European Organization for Nuclear Research, operates a wide set of particle accelerators which rely on critical industrial control systems such as cooling, ventilation, vacuum or cryogenics for the superconducting magnets. These industrial systems are controlled by PLCs (Programmable Logic Controllers) which are themselves programmed in ST (Structured Text) as defined by the IEC 61131-3 standard [17].

One of the main issues of PLC programming is the lack of modern software engineering best practices to guarantee the intended behaviour of the system. While standards such as IEC 61508 [18] give some guidelines and good practices, producing high quality code remains a challenging task. Manual and automated testing are widely-used techniques to test PLC programs and guaranteeing correct behaviour. Even if good results have been achieved using these techniques, not all the problems can be solved. One of them is the difficulty to check safety or liveness properties, e.g. ensuring a forbidden output value combination should never occur.

Formal verification, and more precisely model checking, appears as an adequate technique to guarantee the behaviour of embedded systems in general and PLC programs in particular (c.f. Table 1). However, this technique is still not widely used in industry for three main reasons:

1. it requires a significant effort and an extensive knowledge of the underlying model checker to build the right formal model;
2. it appears that each model checker has its own advantages and disadvantages, which requires ideally to create several formal models to completely validate a single program;
3. real-life systems are generally too large to be automatically analysed by existing model checkers and require reduction techniques to be applied.

In this paper, we propose a general methodology to verify PLC programs by automatically generating formal models for different model checkers out of ST code. Our goal is to perform automatic verification of complex properties coming from real design of these PLC programs. These requirements can be expressed using Computation Tree Logic (CTL) or Lineal Temporal Logic (LTL) expressions. The proposed methodology defines an automata-based formalism used as *intermediate model* (IM) to transform PLC programs written in ST code into different formal models for verification purposes. The transformation is done in two steps: first performing a transformation from the ST code into the IM; and then translating the IM into formal models used as inputs by the model checkers. The IM is an independent representation of the ST code, on which reduction and abstraction techniques can be applied to produce simplified models, manageable by the model checkers in terms of state space. In addition, an automatic generation tool has been developed to implement this methodology. This generation tool hides the difficulty of creating the models out of PLC programs to the control engineers, and allows to perform formal verification using different model checkers.

Related work. While other work have been carried out in the past to apply formal verification to PLC programs, c.f. Table 1, it usually does not address programs written in ST code [4, 22, 6, 7, 2, 8, 19], also they do not provide reduction solutions or they impose strict limitations. Only a few works targeting ST code verification can be found in the literature [16, 21, 20]. However, they either impose restrictions on the ST code such that the approach is inapplicable to real-life systems [16] (e.g. only Boolean variables can be used, or loops are not allowed), or the requirement specifications are limited to assertions due to the underlying methodology [21, 20]. Finally, all the works found in literature target a specific verification tool, and perform transformations for only one type of formal models. The methodology proposed in this paper does not restrict the expressiveness of the ST language allowing

Table 1: Related work

Reference	Input lang.	Verifier	Req. language
[4]	SFC	Cadence SMV	CTL
[22]	SFC	UPPAAL	CTL subset
[4]	timed SFC	UPPAAL	CTL subset
[6]	SFC, IL	Coq	theorems
[7]	SFC, FBD	BIP	—
[2]	FBD	PetriDotNet	CTL
[8]	IL	Cadence SMV	LTL
[19][1]	IL	UPPAAL	CTL subset
[21, 20]	ST	Coq[2]	assertions
[16][3]	ST	NuSMV	—

[1] Only Boolean variables are permitted.
[2] The proposed approach is to convert the ST code into ANSI C first, then translate it to the input of the Coq theorem prover using the Why framework. The verification is done by Coq.
[3] The following limitations apply: only Boolean variables, no iteration statements. The method can be applied for LD and IL too, but it is not presented in [16].

the use wide range of data types (e.g. boolean, integer, float or compound types) or PLC function and function blocks. Also, it supports producing models for different verification tools. In addition, it supports requirement specifications expressed in CTL or LTL, allowing the verification of complex properties.

The rest of the paper is structured as follows: Section 2 describes and justifies the general aspects of the proposed methodology. Section 3 defines the automata-based formalism used as IM and its semantics. The transformation from ST code to the IM is presented in Section 4. Section 5 gives some examples of the transformation rules from the IM to the input language of NuSMV model checker. Section 6 presents experimental results and analysis of the methodology and the tool. Finally, the conclusions and the future work are depicted.

2 Methodology

This section presents the general overview and the motivation of the proposed methodology to automatically produce formal models of PLC-based control systems for verification purposes.

2.1 Overview

The methodology relies on an intermediate model and a set of transformation tools to produce the input models for different verification tools from the "non-formal world" of PLC control systems (see Fig. 1).

In the first step of the methodology, the ST code is parsed, building an Abstract Syntax Tree (AST) which represents the syntax of the PLC code. This AST is transformed to a Control Flow Graph (CFG) which represents the semantics of the code as an automata-based IM. The transformation uses the abstract model of the PLC hardware to be able to represent the *scan cycle*. PLC programs are executed cyclically: at the beginning of each cycle inputs are read and they keep their value during the cycle.

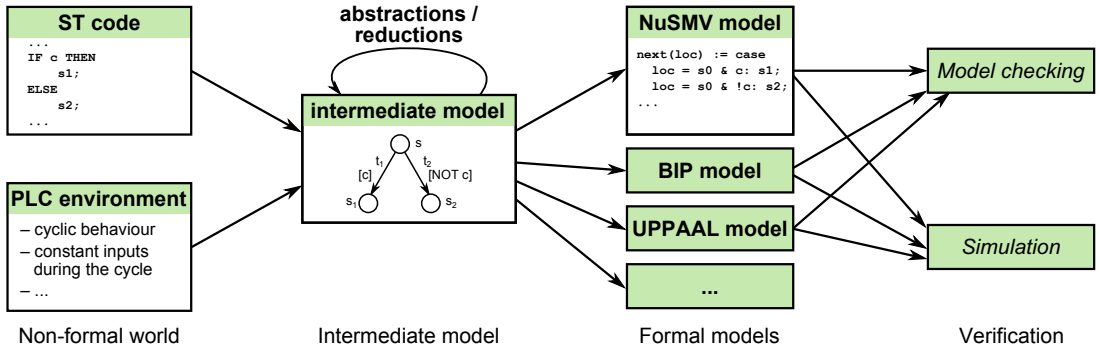


Figure 1: Overview of our approach

The second step of the methodology consists in a transformation from the IM to the specific input language of the verification tools. We designed the IM in a way to be semantically close to the model representation of the different verification tools, therefore these transformations are relatively simple, they just handle the differences of the input formats of the verification tools.

Finally, having these specific models, simulation and verification can be applied to the model of PLC programs.

This paper is focusing on the definition of the IM, the transformation rules from ST code to the IM and the transformation rules from the IM to different model checkers.

2.2 Motivation

Some obvious questions can be arisen after the brief introduction of our methodology. The rest of this section is dedicated to answer them in order to introduce and justify our decisions.

Why introducing an intermediate model? There are multiple reasons to include an intermediate step in our approach, including: *(a)* The transformation rules are split into two parts: "AST to CFG" transformation from the ST code to the IM and "CFG to CFG" transformation from the IM to the different model checker inputs. The "CFG to CFG" transformations are much more simple than the first ones. By decoupling the transformations in two distinctive steps it allows to clearly separate the two roles of the transformation by making them independent one from another (separation of concerns design approach). *(b)* This approach allows us to easily add new model checkers, if their input languages are close to an automata-based formalism. *(c)* Abstraction techniques can be applied to the IM, therefore the performance of the verification can be increased for all the model checkers included in the tool chain.

Why automaton-based intermediate model? Automata-based formalism is a simple formalism but strong enough to model all the features of a PLC control system (Section 4 and 5 shows examples of features in ST code modeled in this formalism). In addition, many verification tools use modeling languages close to this idea, for example the selected tools in this methodology: NuSMV [9], UPPAAL [1] and the BIP framework [3]. Therefore the transformation rules between the IM and the input of NuSMV, UPPAAL, DFinder (from the BIP framework) or any other similar model checkers are simple to implement and the methodology, as well as the tool, can be easily extended.

Why more than one model checker is needed? The existing verification tools provide different advantages and disadvantages in terms of performance, simulation facilities and properties specification. Our goal is not to develop a new verification tool, therefore we wanted to compare them according to this three features and provide the PLC developers the best alternative for our models. For instance, up to now, NuSMV provides better results in terms of verification performance for our current models. NuSMV also supports the full LTL and CTL for the specification properties but it lacks good simulation facilities; UPPAAL provides very good simulation facilities but it supports only a subset of CTL; BIP

provides a language for modeling component-based systems, code generation and simulation facilities. From BIP, model-based automated testing can be applied, some results applied to PLC control systems can be found in [13] and its verification tool for compositional verification (called DFinder) only supports deadlock and safety properties. New and improved verification algorithms and tools are being developed and they can be included here. This strategy makes the methodology independent of a single verification tool.

How to avoid state space explosion in large PLC program models? Automata-based models of PLC programs, as any software model for verification purposes, usually face the problem of huge state space. Several abstraction and reduction techniques can be applied to the IM and all the specific model formats can benefit from these techniques. While the reduction techniques applied to the IM are too complex to be presented in detail in this paper, Section 6 gives an overview of the techniques applied. Readers may refer to [11] for detailed description.

3 Automata-based formalism and its semantics

This section presents the automata-based formalism used as IM. First, the definition of the formalism is presented and then its semantics is introduced.

3.1 Formalism

To represent the PLC programs, we use an automata-based formalism. Here we define a simple automata network model, where independent automata can be defined and synchronized. The formalism presented is similar to the network of timed automata formalism defined in [5], but without explicit logical clock representation and with slightly different interaction (synchronization) semantics. The reason for not using explicit logical clock representation is motivated by the fact that we want to apply the simplest reasonably possible model as IM to make easier to transform the IM into different model checkers language. The strategy for modeling the timing aspects of a PLC is referenced in Section 4.

A *network of automata* is a tuple $N = (A, I)$, where A is a finite set of automata, I is a finite set of interactions.

An *automaton* is a structure $a = (L, T, l_0, V, \underline{Val}_0) \in A$, where $L = \{l_0, l_1, \dots\}$ is a finite set of locations, T is a finite set of guarded (interactive) transitions, $l_0 \in L$ is the initial location of the automaton, $V = \{v_1, \dots, v_m\}$ is the finite set of variables, the value domain for every variable v is $\mathscr{D}_v$, and $\underline{Val}_0 = (Val_{1,0}, \dots, Val_{m,0})$ is the initial value of the variables $(\forall v_i \in V : Val_{i,0} \in \mathscr{D}_{v_i})$.

A *state* of an interactive automaton is a pair $LV_a = (l, \underline{Val})$, where $l \in L$ is the current location and $\underline{Val}$ is the vector of current values of each variable $v \in V$ (in a fixed order).

A *transition* is a tuple $t = (l, g, amt, i, l')$, where $l \in L$ is the source location, $g : (V \to \mathscr{D}_v) \to bool$ is the guard, $amt : (V \to \mathscr{D}_v) \to (V \to \mathscr{D}_v)$ is the memory change (variable assignment), $i \in I \cup \{NONE\}$ is an interaction connected to the transition, and $l' \in L$ is the target location.

An *interaction* is a structure $i = (t, t', amt)$, where $t \in T$ and $t' \in T'$ are two transitions in different automata, and $amt : (V \to \mathscr{D}_v) \to (V \to \mathscr{D}_v)$ is the memory change (variable assignment).

3.2 Semantics

The behaviour of this automata-based formalism can be easily explained as follows: when an automaton is in location l and a transition t goes from l to l', it is enabled if its guard g is satisfied and it has no interactions ($i = NONE$). If this transition t occurs (fires), the location of the automaton will be l' and the variable assignments defined for t will be executed. The transitions joined by an interaction can only fire together and only if both are enabled (synchronous composition).

In the next paragraphs, the previous informally introduced semantics is presented in a formal way. For that purpose, the product-automaton of our IM (with regards to the interactions) is represented as a state transition system (STS).

A (finite) *state transition system* is a $(S_{STS}, s_0, \mathscr{N})$ tuple, where S_{STS} is a finite set of states, $s_0 \subseteq S_{STS}$ is a set of initial states, $\mathscr{N} \subseteq S_{STS} \times S_{STS}$ is a relation over S_{STS} which describes the possible state changes i.e. transitions. In the following, a transition from state s to s' will be marked as $s \rightarrow s'$ to improve readability.

Let $N = (A, I)$ be an interactive automata network (where $A = \{a_1, \ldots, a_n\}$). The *semantics of this automata network* can be defined as a state transition system $(S_{STS}, s_0, \mathscr{N})$, where S_{STS} is the set of states $LV_{a_1} \times \cdots \times LV_{a_n}$, and $s_0 = (l_{0,1}, \underline{Val}_{0,1} \ldots, l_{0,n}, \underline{Val}_{0,n})$ is the initial state. The set of transitions $\mathscr{N} \subseteq S_{STS} \times S_{STS}$ is constructed in the following way:

- For every transition $t = (l, g, amt, int, l') \in T$ in an automaton a_i, where $int = NONE$, for every $(\underline{Val}, \underline{Val}^*) \in amt$: if $g(\underline{Val}) = true$, then add:

$$(lv_1, \ldots, lv_{i-1}, (l, \underline{Val}), lv_{i+1}, \ldots, lv_n) \rightarrow (lv_1, \ldots, lv_{i-1}, (l', \underline{Val}^*), lv_{i+1}, \ldots, lv_n)$$

  ($\forall lv_j \in LV_{A_j}$, where $j \neq i$).

- For every transition $t = (l, g, amt, int, k) \in T$ in automaton a_i, where $int = (t, t', iamt)$ and $t' = (l', g', amt', int, k')$ in automaton a_j, for every $(\underline{Val}, \underline{Val}^*) \in amt$, for every $(\underline{Val}', \underline{Val}^{*\prime}) \in amt'$, for every $(\underline{Val}^*, \underline{Val}^{**}) \in iamt$, and for every $(\underline{Val}^{*\prime}, \underline{Val}^{**\prime}) \in iamt$: if $g(\underline{Val}) = true$ and $g'(\underline{Val}') = true$, then add:

$$(lv_1, \ldots, lv_{i-1}, (l, \underline{Val}), \ldots, (l', \underline{Val}'), lv_{j+1}, \ldots) \rightarrow (lv_1, \ldots, lv_{i-1}, (k, \underline{Val}^{**}), \ldots, (k', \underline{Val}^{**\prime}), lv_{j+1}, \ldots)$$

  ($\forall lv_x \in LV_{A_x}$, where $x \neq i$ and $x \neq j$).

An example of this approach to describe the automata semantics as STS is shown in Fig. 2.

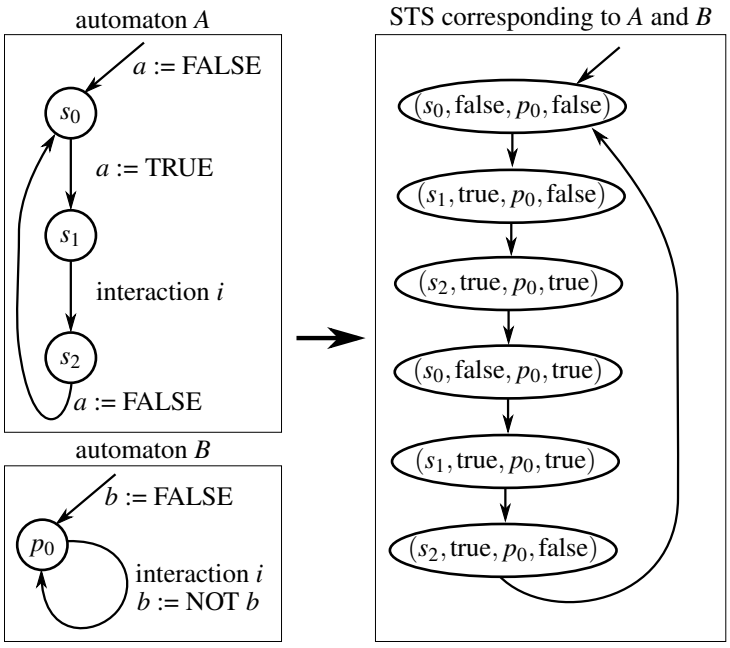


Figure 2: Example of representing interactive an automata network as a state transition system (STS)

4 Transformation from ST to automata

This section presents the "AST to CFG" transformation rules between the ST code and the IM. ST is a high-level programming language comprising a list of statements. A statement can be considered as the smallest standalone element of a program and it can contain other components (e.g. expressions). There are different kinds of statements like conditional branches, loops, variable assignments and calls to subroutines. An expression is a group of symbols that represents a value.

For every statement $stmt$, let $n(stmt)$ be the next statement after $stmt$. As we model cyclic PLC programs, the last statement in the ST code is followed by the first one. Furthermore, for a statement list sl let $first(sl)$ be the first statement of the list.

The general representation of the ST program in our automata-based formalism is as follows:

Rule PLC1 For every ST *function block instance fb*, there is exactly one automaton in the model, which is denoted as $F_{FB}(fb)$.

Rule PLC2 For every ST *statement stmt* in function block instance *fb*, there is at least one corresponding location marked as $F_S(stmt)$ in the automaton $F_{FB}(fb)$.

Rule PLC3 For every ST *variable v* in function block instance *fb*, there is exactly one corresponding variable $F_V(v)$ in the automaton $F_{FB}(fb)$.

We add also statement-specific parts to the automata. Here we present the rules corresponding to variable assignments, conditional statements and function calls. These rules are illustrated on Fig. 3.

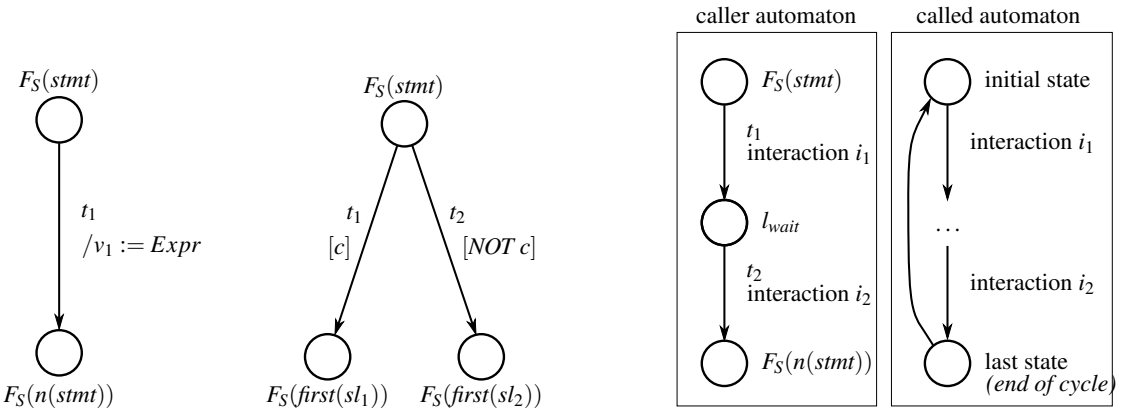
Rule PLC4 For every *variable assignment* $stmt = \langle v_i := Expr \rangle$ in function block instance *fb*, we add a transition to $F_{FB}(fb)$ which goes from $F_S(stmt)$ to $F_S(n(stmt))$ which has no guard and no interaction, and it assigns $Expr$ to the variable $F_V(v_i)$ and does not modify the other variables. (Formally: $t = (F_S(stmt), TRUE, a, NONE, F_S(n(stmt)))$, where assignment a assigns $Expr$ to the variable $F_V(v_i)$ and does not modify the other variables.)

Rule PLC5 For every *conditional statement* $stmt = \langle IF\ c\ THEN\ sl_1\ ELSE\ sl_2\ END\_IF \rangle$ in function block instance *fb*, we add two transitions (t_1 and t_2) to automaton $F_{FB}(fb)$:

- The transition t_1 goes from $F_S(stmt)$ to $F_S(first(sl_1))$, it has no assignments and no interactions, and it has a guard c. (Formally: $t_1 = (F_S(stmt), c, a_{identity}, NONE, F_S(first(sl_1)))$)
- The transition t_2 goes from $F_S(stmt)$ to $F_S(first(sl_2))$, it has no assignments and no interactions, and it has a guard $NOT\ c$. (Formally: $t_2 = (F_S(stmt), NOT\ c, a_{identity}, NONE, F_S(first(sl_2)))$)

Rule PLC6 For every *function call* $stmt = \langle r := Func(p_1 := Expr_1, p_2 := Expr_2, \ldots) \rangle$ in function block instance *fb*, we add the following elements:

- A new location l_{wait} is added to $F_{FB}(fb)$. It represents the state when *fb* is waiting for the end of the function call. (For every function call, we add a separate l_{wait} location.)
- A transition t_1 is added to $F_{FB}(fb)$, which has no assignment, no guard and goes from $F_S(stmt)$ to l_{wait}.
- A transition t_2 is added to $F_{FB}(fb)$, which has no assignment, no guard and goes from l_{wait} to $F_S(n(stmt))$.
- An interaction i_1 is added to the automata network. It connects transition t_1 with the first transition of $F_{FB}(Func)$ and assigns the function call parameters to the corresponding variables in $F_{FB}(Func)$. (It assigns $Expr_1$ to $F_V(p_1)$, etc.)



(a) Corresponding automaton fragment for variable assignment $v_1 := Expr$

(b) Corresponding automaton fragment for conditional statement *IF c THEN* sl_1 *ELSE* sl_2 *END_IF*

(c) Corresponding automaton fragment for function call $r := Func(p_1 := Expr_1, p_2 := Expr_2, \dots)$

Figure 3: Transformation rules from ST code to automata

– An interaction i_2 is added to the automata network. It connects the last transition (which goes to the initial state) of $F_{FB}(Func)$ with transition t_2 and assigns the return values of the function call to the corresponding variable (variable $F_V(r)$) in $F_{FB}(fb)$. It also assigns the corresponding values to the output variables.

In addition to these rules, PLC code often contains timers and modeling these timing aspects is a challenging task and it highly increases the state space of the models. We have described the modeling of timers in [12]. In this paper, however, we concentrate on our general modeling methodology.

Finally, this transformation does not support recursive function calls. However, according to IEC 61131 standard, functions "shall not be recursive" [17].

5 Transformation from IM to specific tools

This section explains some relevant examples of the "CFG to CFG" transformations from the IM to the specific input language of the selected model checkers.

We developed transformation rules from IM to the input model language of NuSMV, BIP and UPPAAL. The three transformations are similar and as a matter of illustration, only the transformation from IM into NuSMV is presented.

Rule PLC1 For the *automata network*, a new module *main* is created. This will contain all the automaton instances. It contains also a variable *interaction*, whose domain is the set of possible interactions in the automata network ($I = \{i_1, i_2, \dots\}$) and the value *NONE*. This variable and the *main* module are passed as parameters to every module instance to be able to access them.

Rule PLC2 For every *automaton a*, a new module *a* is created in the NuSMV model with one single instance *inst_a*. This module will contain a variable *loc* which stores the current location of the automaton. The domain of this variable is the set of possible locations in automaton *a* (i.e., $L = \{l_0, l_1, \dots\}$). The default value of the variable *loc* will be the initial location l_0. The corresponding NuSMV code fragment can be seen on Fig. 4.

Rule PLC3 For every variable *v* in automaton *a*, a new variable *v* is created in the NuSMV model *a*. Each variable will have a next-value assignment statement in the assignment block of the module. If a variable is not modified by an assignment explicitly, it will keep its value, as it can be seen on Fig. 6.

```
MODULE a(main, interaction)
  VAR
    loc : {l0, l1, l2, ...};
    ...
  ASSIGN
    init(loc) := l0;
    ...
MODULE main
  VAR
    inst_a : a(self, interaction);
```

Figure 4: Representation of automaton a in NuSMV

Rule PLC4 For every *transition* $t = (l_1, g, amt, i, l_2)$ in automaton a, a new assignment rule is added for the corresponding *loc* variable. It will express that if the current location is l_1, the next location will be l_2, if guard g is true and interaction i is enabled. If there is no interaction or guard connected to transition t, the corresponding condition can be omitted. This transformation can be seen on Fig. 5.

If the transition t contains *variable assignment* $v := Expr$ for variable v, the next-value assignment corresponding to v is extended too, as it can be seen on Fig. 6.

Rule PLC5 For each *interaction* i_1 connecting transition t_1 which goes from location l_1 in automaton a and transition t_2 with source l_2 in automaton b, an invariant is added to the model, as it can be seen on Fig. 7.

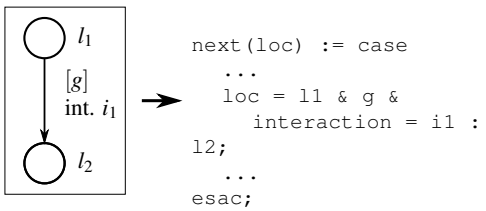


Figure 5: Representation of transition $t = (l_1, g, amt, i, l_2)$

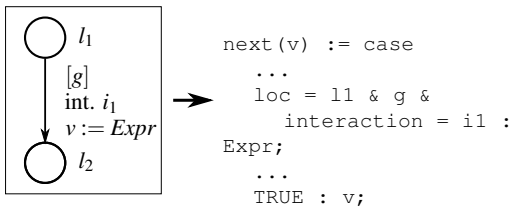


Figure 6: Representation of variable assignment $v := Expr$

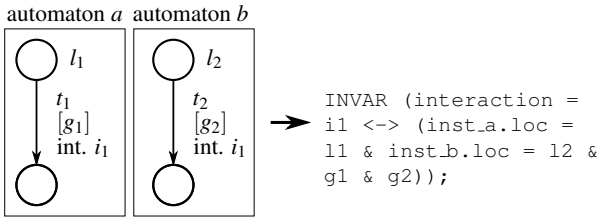


Figure 7: Invariant representing interaction i_1

Figure 8 shows an example of a small ST code, the corresponding IM and NuSMV code. The ST code contains a single function block which implements a counter without any function calls. The corresponding IM have a single automaton with no interactions. The NuSMV model generated from IM contains two modules: the module *main* for the network and the module *counter* for the single automaton.

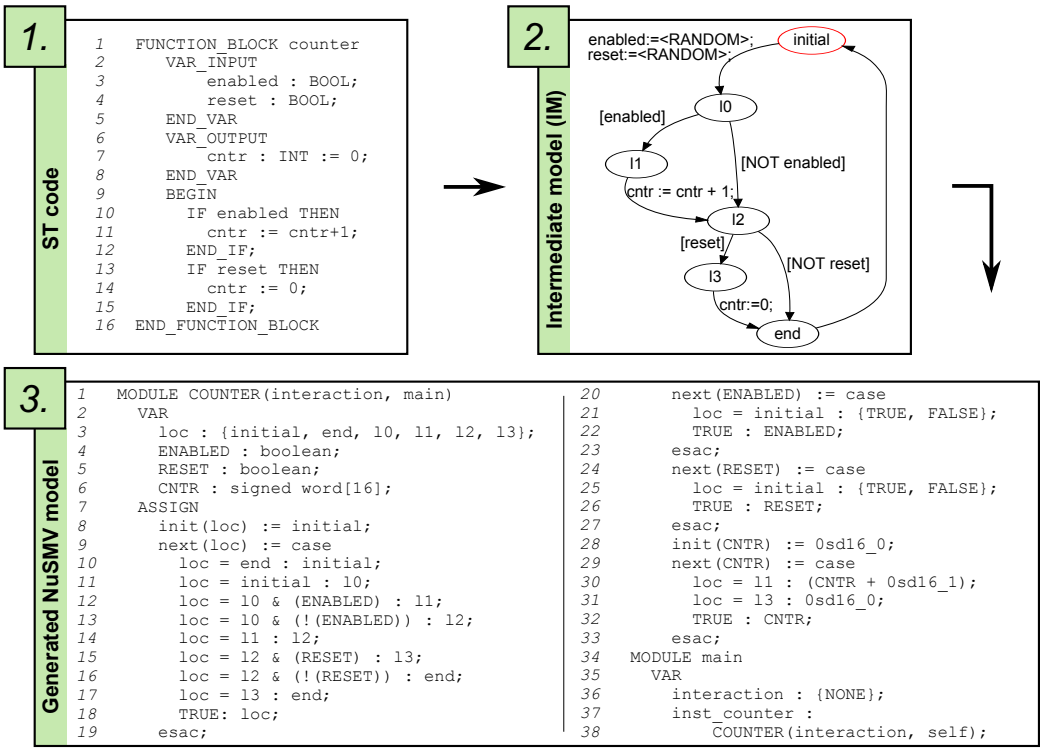


Figure 8: Example ST code, the corresponding IM and the corresponding NuSMV code

The representation of statements can be also observed on this simple case. For example, in line 11 of the ST code, the variable *cntr* is incremented if the condition *enabled* is true. In the IM this statement is represented as a transition between *l1* and *l2*. In the NuSMV model, the transition is represented by two lines of code: line 14 expresses the modification of the location variable, line 30 describes the modification of the variable *cntr*.

6 Experimental results and analysis

This section presents the tool supporting the methodology as well as its application on a real case study.

6.1 Implementation

A transformation tool supporting the presented methodology has been developed using EMF (Eclipse Modeling Framework) and Xtext technologies. The tool implements the relevant parts of the ST grammar and the automata-based IM as an EMF metamodel. The transformation rules have been implemented using Xtend and Java. The tool generates NuSMV, BIP and UPPAAL models from the ST source code.

Models of real-life programs with a significant number of variables, functions, etc. often have huge state spaces. To address this problem, we apply reduction techniques at the IM level. Thus, all final models benefit from them. This is not in the scope of this paper and the details of the reduction techniques can be found in [11]. However we briefly introduce the applied techniques to support the results. Two types of reductions have been implemented:

– *General rule-based reductions* are used to simplify the models. These reductions can be used for every kind of PLC code. For example, variable assignments on succeeding transitions can be merged if they are not affecting each other, because the value of the variables are not checked during the execution of the PLC cycle. More than 20 rules have been introduced to simplify and reduce the model. These reductions are applied iteratively.

– When the property to verify is provided, we apply the *Cone of Influence* (COI) reduction [10], which consists in removing all variables that do not affect the property. As it is depending on the requirement to be checked, its effect on the size of the state space highly varies, but for many examples, we observe a large state space reduction.

6.2 Applicability

PLC control systems developed with the UNICOS framework [15] are used as case study. UNICOS contains a library of base objects representing common industrial control instrumentation (e.g. sensors, actuators, subsystems). These objects are represented as function blocks in the PLC code, using the ST language and can invoke different function blocks on the PLC. Specifically the case study uses the ST code targeting Siemens PLCs.

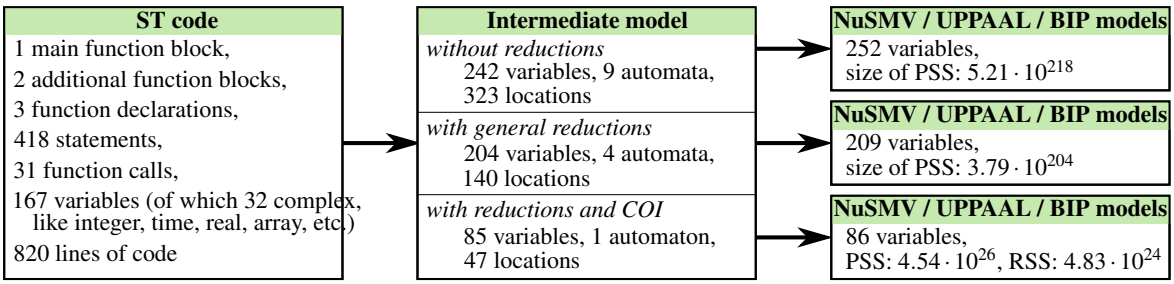


Figure 9: Measurements for the OnOff UNICOS object

As an example for this paper, the OnOff object has been chosen. This object can represent any binary-like process equipment (actuators) driven by digital signals (e.g. valves, heaters, motors, etc.). This is not the most complex module in the UNICOS library, but its size and complexity are representative (60 input and 62 output variables). Generation and reduction time together of the NuSMV, BIP or UPPAAL models is under 1 seconds.

Fig. 9 shows some key metrics for the OnOff UNICOS object. As it can be seen, the ST code contains a significant number of statements and variables of different nature. The corresponding IM and generated models have a huge potential state space (PSS) before reductions. After applying general reductions and COI the size of the models were reduced significantly. Note that the COI reduction depends on the given requirement. In this case, the requirement was a safety property containing 9 variables depending on 79 other variables. The final reduction of the PSS was from $5.21 \cdot 10^{218}$ states to $4.54 \cdot 10^{26}$, while the size of the reachable state space (RSS) became $4.83 \cdot 10^{24}$. The final model was reduced enough for verification and simulation using NuSMV, UPPAAL and BIP, showing the necessity of our work. A case study with verification results of complex properties can be found in [14].

7 Conclusion and future Work

The paper presents an approach to create formal models of ST PLC programs for verification purposes. The proposed methodology is generic and can be widely applied to ST PLC programs. We implemented the proposed transformations in an automatic generation tool based on EMF metamodels. Currently this tool produces formal models for NuSMV, UPPAAL and BIP from ST code.

The results showed that it is possible to represent ST code with an automaton-based formalism. It has been also shown that an intermediate model can make the transformation simpler, and it facilitates to extend the approach for other IEC 61131 input languages or model formalisms. In the proposed methodology, various abstraction and reduction techniques can be applied to reduce the state space, allowing the verification of the generated models.

Currently the tool does not support all the software blocks provided by the PLC operating system (e.g. built-in function blocks), as their ST source code is not available. The transformation tool does not take into account anything that is not defined in the given source code, except some general assumptions about PLCs, like the cyclic behaviour or time representation.

Future plans for this project are the integration of the tool and methodology in the UNICOS development process at CERN. In addition, extending and optimizing the abstraction techniques and the proof of the transformation to guarantee its correctness, are ongoing work.

References

[1] Tobias Amnell et al. UPPAAL – now, next, and future. In *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 99–124. Springer, 2001.

[2] Tamás Bartha, András Vörös, Attila Jámbor, and Dániel Darvas. Verification of an industrial safety function using coloured Petri nets and model checking. In *Proceedings of the 14th International Conference on MITIP 2012*, pages 472–485. Hungarian Academy of Sciences, Computer and Automation Research Institute, 2012.

[3] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28:41–48, 2011.

[4] Nanette Bauer, Sebastian Engell, Ralf Huuck, Sven Lohmann, Ben Lukoschus, Manuel Remelhe, and Olaf Stursberg. Verification of PLC programs given as sequential function charts. In Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Groe-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 517–540. Springer, 2004.

[5] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, volume 3185 of *Lecture Notes on Computer Science*, pages 200–237. Springer Verlag, 2004.

[6] Jan Olaf Blech and Sidi Ould Biha. Verification of PLC properties based on formal semantics in Coq. In *Software Engineering and Formal Methods*, volume 7041 of *Lecture Notes in Computer Science*, pages 58–73. Springer, 2011.

[7] Jan Olaf Blech, Anton Hattendorf, and Jia Huang. An invariant preserving transformation for PLC models. In *Proc. of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 63–71, 2011.

[8] Géraud Canet, Sandrine Couffin, Jean-Jacques Lesage, Antoine Petit, and Philippe Schnoebelen. Towards the automatic verification of PLC programs written in Instruction List. In *Proc. of the IEEE Int. Conf. on Systems, Man and Cybernetics*, pages 2449–2454. Argos Press, 2000.

[9] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim G. Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364, 2002.

[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[11] Dániel Darvas, Borja Fernández Adiego, András Vörös, Tamás Bartha, Enrique Blanco Viñuela, and Víctor M. González Suárez. Formal verification of complex properties on PLC programs. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, volume 8461 of *LNCS*, pages 284–299. Springer, 2014.

[12] B. Fernández Adiego, D. Darvas, E. Blanco Viñuela, J.-C. Tournier, V. M. González Suárez, and

J. O. Blech. Modelling and formal verification of timing aspects in large PLC programs. In *Proc. of IFAC World Congress*, 2014. To appear.

[13] Borja Fernandez Adiego, Enrique Blanco Vinuela, Jean-Charles Tournier, Víctor M. Suárez González, and Simon Bliudze. Model-based automated testing of critical PLC programs. In *11th IEEE International Conference on Industrial Informatics*, pages 722–727, 2013.

[14] Borja Fernández Adiego, Dániel Darvas, Jean-Charles Tournier, Enrique Blanco Viñuela, and Víctor M. González Suárez. Bringing automated model checking to PLC program development – A CERN case study. In Jean-Jacques Lesage, Jean-Marc Faure, José E. Ribiero Cury, and Bengt Lennartson, editors, *Preprints of the 12th International Workshop on Discrete Event Systems*, pages 394–399, 2014.

[15] Philippe Gayet and Renaud Barillère. UNICOS a framework to build industry like control systems: Principles & methodology. In *10th ICALEPCS*, 2005.

[16] Vincent Gourcuff, Olivier De Smet, and Jean-Marc Faure. Efficient representation for formal verification of PLC programs. In *8th Int. Workshop on Discrete Event Systems*, pages 182–187, 2006.

[17] IEC 61131: Programming languages for programmable logic controllers.

[18] IEC 61508: Functional safety of electrical / electronic / programmable electronic safety-related systems.

[19] Angelika Mader and Hanno Wupper. Timed automaton models for simple programmable logic controllers. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 106–113. IEEE Comp. Soc. Press, 1999.

[20] Jan Sadolewski. Automated conversion of ST control programs to Why for verification purposes. In *Federated Conference on Computer Science and Information Systems*, pages 849–854, 2011.

[21] Jan Sadolewski. Conversion of ST control programs to ANSI C for verification purposes. *e-Informatica*, 5(1):65–76, 2011.

[22] Cleber A. Sarmento, José R. Silva, Paulo E. Miyagi, and Diolino J. Santos Filho. Modeling of programs and its verification for programmable logic controllers. In *17th IFAC World Congress*, 2008.