# Applying model checking to industrial-sized PLC programs

Borja Fernández Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, *Member, IEEE,* Simon Bliudze, Jan Olaf Blech, *Member, IEEE,* and Víctor Manuel González Suárez

***Abstract*—Programmable logic controllers (PLCs) are embedded computers widely used in industrial control systems. Ensuring that a PLC software complies with its specification is a challenging task. Formal verification has become a recommended practice to ensure the correctness of safety-critical software but is still underused in industry due to the complexity of building and managing formal models of real applications. In this paper, we propose a general methodology to perform automated model checking of complex properties expressed in temporal logics (e.g., CTL, LTL) on PLC programs. This methodology is based on an intermediate model (IM), meant to transform PLC programs written in various standard languages (ST, SFC, etc.) to different modeling languages of verification tools. We present the syntax and semantics of the IM and the transformation rules of the ST and SFC languages to the nuXmv model checker passing through the intermediate model. Finally, two real cases studies of CERN PLC programs, written mainly in the ST language, are presented to illustrate and validate the proposed approach.**

***Index Terms*—PLC, IEC 61131, modeling, automata, verification, model checking, nuXmv.**

## I. Introduction

DEVELOPING safe and robust PLC (Programmable Logic Controller) based control systems is a challenging task for control system engineers. One of the biggest difficulties is to ensure that the PLC program fulfills the system specification. Some standards, such as IEC 61508 [1] give some guidelines and good practices, but this task remains challenging. Many different techniques are widely applied in industry meant to check PLC programs, e.g., manual and automated testing or simulation facilities. However, they still present some significant problems, like the difficulty to check safety or liveness properties, e.g., ensuring that a forbidden output value combination never occurs. Formal verification techniques can handle these problems, but bring other challenges to the control engineers such as the construction of the formal models and the state space explosion when applied to real-life software applications.

Manuscript received June 30, 2014; revised June 29, 2015; accepted for publication September 28, 2015.

B. Fernández, D. Darvas, E. Blanco, and J-C. Tournier are with CERN (European Organization for Nuclear Research), Switzerland.
S. Bliudze is with Ecole polytechnique fédérale de Lausanne, Switzerland.
J. O. Blech is with RMIT University Melbourne, Australia.
V. M. González is with University of Oviedo, Spain.

### *A. Contribution*

Our motivation is to find software faults (bugs) by applying automated formal verification of complex properties—expressed in temporal logic—to real-life PLC control systems developed at CERN, the European Organization for Nuclear Research. We provide a general methodology for automatic creation and verification of formal models from code written in different PLC languages, which also handles the state space explosion problem. Although the main focus of this paper is on the transformation of PLC programs into formal models, we provide a description of the full methodology and illustrate it on two real-life examples. The specific contributions of this paper are the following:

1) We present the formal transformation rules from ST (Structured Text) and SFC (Sequential Function Chart)—the two most used languages in CERN PLC control systems—to IM (Intermediate Model) and give an overview of the transformation from IM to one of the selected model checker modeling languages: nuXmv. This is presented in Section IV.
2) The methods proposed in our previous work are extended to be applicable to large, industrial-size PLC programs. The methodology has been applied to real-life systems at CERN. The experimental results are discussed in Section V.

This paper presents an extension of a previous work meant to bring formal verification to the industrial automation community. A first method [2] was proposed to model various software components of PLC programs developed at CERN, using the BIP framework exclusively. A first version of the transformation rules from ST code to the NuSMV modeling language is described in [3]. Compared to this previous work, the present paper *(a)* extends and refines the rules presented previously, *(b)* encompasses other languages than ST, and *(c)* presents an application of the approach to a real-life case study. The model reduction techniques and the representation of time-related behavior is not in the main scope of this paper, but the methods used in [4], [5] can be applied here as well.

### *B. Related Work*

Although application of formal methods to PLC software has been extensively studied in the existing literature [6]–[25], none of the described methods achieves the goals stated above.

In [6], one can find a fairly complete survey and classification of PLC verification methods. Using this classification our

*Authors' manuscript. The final version was released in IEEE Transactions on Industrial Informatics, Vol. 11, Issue 6, pp. 1400-1410, 2015. It can be obtained from IEEE Xplore: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7295624. DOI:* 10.1109/TII.2015.2489184

method is in the "M-A-M" group, meaning that it is a model-based approach, relies on automata and model checking. A recent survey [26] introduces different classifications for model checking methods applied to the PLC domain. Our method covers many different classes, e.g., it covers multiple PLC languages and multiple system sizes. Furthermore, it aims to be fully automated. The application area is also broad, as PLCs are used for many purposes at CERN.

Some commercial tools, e.g., SCADE from Esterel Technologies[1] provide solutions for the generation of safe PLC programs, where certified PLC code is automatically generated from a formal specification. This approach does not fit the practical industrial requirements, as often already existing PLC programs have to be verified.

In the academic literature, some of the work only targets the modeling of PLCs without providing a verification solution [7]. Other authors apply formal verification, but only for small examples, without discussing the reduction of the models that is inevitable for verifying industrial-sized programs [8]–[16]. Many papers do not address the generation of the model from the PLC program or limit themselves to explaining the high-level principles [10]–[20]. Finally, most of the work targets a single PLC language, with just a few approaches handling multiple ones (e.g., [21], [22]).

In [18], CEGAR (counterexample-guided abstraction refinement) is applied to models of PLC programs, but only ACTL (Computation Tree Logic with only universal path quantifiers) formalism is supported for property specifications. The work in [24] uses CEGAR too, but only for reachability analysis. In [23], the authors introduce powerful reduction methods applied to IL (Instruction List) code. Although this approach could be extended to other languages, reliance on SMT (Satisfiability Modulo Theories) solvers restricts its applicability to safety requirements.

Some work targets specifically the verification of ST programs [21], [22]. However, the methods described in [21] restrict the requirements to assertions, which have smaller expressiveness than LTL (Linear Temporal Logic) or CTL (Computation Tree Logic). Although powerful reduction techniques are proposed in [22], they also have strong limitations. For instance, programs can only contain non-Boolean variables and no loops. Applicability of this method for industrial-sized applications at CERN is questionable, since these would contain highly complex Boolean expressions.

The approach based on intermediate model, proposed in this paper is new in the PLC domain, however approaches using similar verification techniques have been applied in other domains [27], [28].

The rest of the paper is structured as follows: Section II presents a general description of PLCs. Section III is dedicated to an overview of the methodology and the applied intermediate model. Section IV discusses the transformation from the ST and SFC languages to IM and gives a high-level overview of the transformations from IM to nuXmv and of the reduction techniques applied to IM. Section V presents experimental results obtained by applying our methodology to CERN control systems. Finally, in Section VI, we discuss the presented results and possible directions for future work.

[1]http://www.esterel-technologies.com/products/scade-suite/

## II. Programmable Logic Controllers

This section presents the PLC concepts necessary to justify the proposed modeling strategy. PLC is a widely-used programmable electronic device designed for controlling industrial processes. It mainly consists of a processing unit and input/output modules to acquire and act with sensors and actuators of the process. Even though the architecture and programming of PLCs is defined in the IEC 61131 standard [29], there are minor differences in the implementation of different manufacturers. In this work, we focus on Siemens PLCs, since these are among the most widely used in the industry and, in particular, at CERN. However, the proposed methodology can be applied to PLCs produced by other manufacturers with only minor adaptation of the transformation rules, necessary to accommodate the variations of PLC programming languages.

*a) Execution scheme:* The main particularity of the PLC is its cyclic execution scheme. It consists of three main steps: (1) reading the input from periphery to the memory, (2) executing the user program that reads and modifies the memory contents, (3) writing the values to the output periphery. The cyclic execution can be interrupted if an event (e.g., timer, hardware event, hardware error) triggers the execution of an interrupt handler. Interrupts are preemptive; they are assigned to priority classes at compilation time.

*b) Program blocks:* In Siemens PLCs, several kinds of program blocks are defined for various purposes [30].

- A *function* (FC) is a piece of executable code with input, output, and temporary variables. The variables are dynamically stored on a stack and they are not retained after the execution of the function.
- An *organization block* (OB) is a special function called by the system. OBs are the entry points of the user code. The main program and the interrupt handlers are implemented as OBs.
- A *data block* (DB) is a group of static variables that can be accessed globally in the program. These variables are stored permanently. A data block does not contain any executable code.
- A *function block* (FB) is a piece of executable code with input, output, static, and temporary variables. An FB can have several instances and each instance has a separate *instance data block* that stores its non-temporary variables. Thus, these variables can be accessed globally, even before or after the execution of the FB. The temporary variables are stored on a stack, as the variables of an FC.

*c) Programming:* PLCs provide several standard programming languages. Five languages are defined in the IEC 61131-3 standard [29]: ST (Structured Text), SFC (Sequential Function Chart), Ladder, FBD (Function Block Diagram), and IL (Instruction List). A PLC programmer can chose one or several of these languages, depending on the characteristics of the application, to build the PLC code.

The prevalent language at CERN is ST. However, SFC and IL are also used. **IL** is a low-level language, syntactically similar to assembly. **SFC** is a graphical programing language based on finite-state machines (FSMs), described using *steps* (states) and *transitions*. Two different kinds of branches are defined: *alternative branches* (where at most one of the branches can contain active steps) and *simultaneous branches* (where each branch contains an active step, or none of them). This formalism is similar to the safe Petri nets, but the semantics is different: the enabled transitions are evaluated once per call and then only these transitions can fire. If a transition becomes enabled due to a firing, it can fire only on the next call of the SFC. Also, steps can have associated actions, such as variable assignments. This language is useful when part of the PLC program can be represented conveniently as a finite-state machine (FSM). **ST** is a high-level language that is syntactically similar to Pascal.

In this paper we target ST and SFC as source languages, more precisely the languages corresponding to them in the Siemens implementations: SCL (Structured Control Language) and S7-GRAPH/SFC. The Siemens implementation follows the IEC 61131 standard as stated by the PLCOpen organization[2] and [31], but there are small syntactic differences between the standard languages and their implementation. SCL language can be used to describe all kinds of program blocks mentioned previously, while SFC can only represent an FB.

Programs written in any of the above languages are compiled into a common byte-code representation, called MC7, which is then transferred to the PLC. Based on our experience, we assume that the MC7 instructions are atomic and they cannot be interrupted. A single ST or SFC statement may correspond to several MC7 instructions, thus it is possible to interrupt an ST or SFC statement.

## III. Modeling and Verification Approach

### A. Methodology Overview

We propose a general methodology for applying automated formal verification to any PLC program written in one of the PLC languages. To support multiple PLC languages, a valid solution could be to first translate them to IL or to machine code, and then only this single, low-level language has to be targeted by verification. While this method can be general, it can cause some information loss. For example, evaluation of an arithmetic expression that could be represented both in the high-level PLC language and in the model checker input language, will be split into several instructions in IL, making the reductions more difficult and the model checkers less efficient.

Instead, the methodology presented here is based on the *intermediate model* (IM) formalism designed for verification purposes (not for machine execution as IL). Each language is translated individually to IM. In this way we can benefit from the higher level inputs (ST vs. IL), that generally provide more information and can be reduced more efficiently. The methodology contains a set of rules which can transform automatically PLC code in different modeling languages passing through IM.

[2]http://www.plcopen.org/pages/tc3_certification/certified_products/

Furthermore, this intermediate step allows us to compare the different model-checking tools in terms of verification performance, simulation facilities and properties specification. More importantly, as each verification tool has different strengths and purposes, we can use the appropriate tool based on our current needs. Currently, translations to the NuSMV/nuXmv, UPPAAL, and BIP verification tools are included in our methodology. IM is based on automata and allows us to extend our methodology with any verification tool which has a similar modeling language (e.g., SAL, Cadence SMV, LTSmin).

The proposed approach consists of the following steps (see Fig. 1):

1) The starting point is the source code of the PLC program and the formalized requirements coming from an informal specification. Using the knowledge of the PLC execution scheme, the PLC code is automatically transformed to IM. This transformation is defined by a set of formal rules presented in Section IV.
2) Several automatic reduction and abstraction techniques are then applied to the generated model, depending on the requirement to be verified.
3) The reduced model is automatically translated to external modeling languages, used by the verification tools.
4) The resulting external models can be formally verified using such tools as nuXmv or UPPAAL. Other tools (e.g., BIP) provide simulation and code generation facilities, which can be useful for PLC developers.
5) Counterexamples produced by model checkers allow PLC developers to analyze the results in order to confirm the presence of bugs in the system or refine the models.

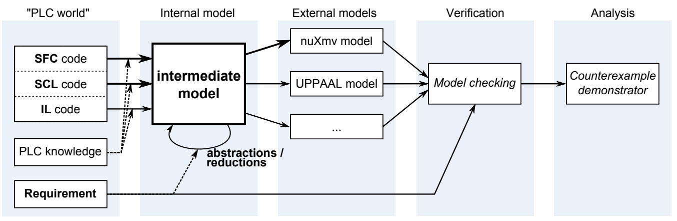


Figure 1. Overview of our approach

### B. Intermediate Model

This section describes briefly the syntax and semantics of IM—our automata-based formalism used to represent the PLC programs. We define a simple automata network model consisting of synchronized automata.

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

An *automaton* is a structure $a = (L, T, l_0, V_a, \underline{\mathit{Val}}_0) \in A$, where $L = \{l_0, l_1, \dots\}$ is a finite set of locations, $T$ is a finite set of guarded transitions, $l_0 \in L$ is the initial location of the automaton, $V_a = \{v_1, \dots, v_m\}$ is a finite set of variables, and $\underline{\mathit{Val}}_0 = (\mathit{Val}_{1,0}, \dots, \mathit{Val}_{m,0})$ is the initial value of the variables.

Let $\hat{V}$ be the set of all variables in the network of automata $N$, i.e., $\hat{V} = \bigcup_{a \in A} V_a$. ($\forall a, b \in A : V_a \cap V_b = \emptyset$ if $a \neq b$)

A *transition* is a tuple $t = (l, g, amt, i, l')$, where $l \in L$ is the source location, $g$ is a logical expression on variables of $\hat{V}$ that is the guard, $amt$ is the memory change (variable assignment, i.e., a function that defines the new values of the variables in $\hat{V}$), $i \in I \cup \{\mathit{NONE}\}$ is a synchronization attached to the transition, and $l' \in L$ is the target location.

A *synchronization* is a pair $i = (t, t')$, where $t \in T$ and $t' \in T'$ are two synchronized transitions in different automata. The variable assignments attached to the transitions $t$ and $t'$ should not use the same variables. This composition operation is restrictive, but sufficient to model PLC programs, as synchronizations will only represent function calls.

The operational semantics of this automata-based formalism can be informally explained as follows: a transition $t = (l, g, amt, i, l')$ from the current location $l$ of an automaton is enabled if $g$ is satisfied and either $t$ has no synchronization attached, i.e., $i = \mathit{NONE}$, or $i = (t, t')$ and the transition $t'$ is also enabled. In the former case, $t$ can fire alone; in the latter case, both $t$ and $t'$ have to fire simultaneously. Each execution step consists in firing one transition or simultaneous firing of two synchronized ones. Upon firing of a transition $t$ as above, $l'$ becomes the new current location of the corresponding automaton and the new values of variables $\hat{V}$ are set using the previous values and the variable assignment $amt$.

## IV. Model Transformations

This section describes in detail the most relevant transformation rules from SCL and SFC to IM[3]. Some of these rules are generic and apply to all PLC languages, the rest are specific to SCL or SFC. In addition, a high-level description of the reduction techniques applied to IM models and the transformation from IM to nuXmv are presented. Also, the main ideas of the tool implementing the methodology and some examples are discussed. This section extends and generalizes the previous work [3].

### A. General PLC to IM Transformation

The transformation rules are presented hierarchically from high-level to low-level rules.

**Rule PLC1 (Multiple concurrent code blocks)** PLC programs are composed by the main program (i.e., OB1 in Siemens PLCs), which is executed cyclically, and the interrupt handlers.

*Assumption 1.* Interrupting blocks and the interrupted blocks should use disjoint set of variables. This is a reasonable assumption, and it can be validated by existing static analysis techniques. According to our experience, different OBs usually use different variables. Furthermore, a high level of concurrency is rare in PLC programs.

Having this assumption, instead of modeling the interrupts in a preemptive manner, we model them with non-preemptive semantics: the model of the PLC scheduler consists in the main program being executed at every cycle, whereas one or several interrupts can be executed non-deterministically at the end of the PLC cycle.

**Rule PLC2 (FC)** This rule translates functions into IM. An OB can be considered as a special FC that is invoked by the operating system, thus this rule also applies to OBs.

*Assumption 2.* Recursion is not allowed, i.e., no FC or FB can directly or indirectly generate a call to itself. This assumption is consistent with the IEC 61131 standard [29]. However, Siemens PLCs allows the use of recursion with some restrictions even if it is not recommended. Recursion can be statically detected by building the call graph of a program and checking whether it contains cycles. Thus we can assume that variables of a function are stored at most once on the stack.

For each function *Func*, we create an automaton $A_{\mathit{Func}}$. The locations, transitions and initial location of this automaton are generated using the rules presented below. For each variable defined in *Func* we create a corresponding variable in $A_{\mathit{Func}}$. If the return type of the function is different from `void`, a special output variable called *RET_VAL* is also added to the automaton. $A_{\mathit{Func}}$ contains at least the initial location *init*, the final location *end* and the transition $t_{end}$ from *end* to *init*.

**Rule PLC3 (FB instance)** This rule translates FB instances into IM. Assumption 2 also applies here.

For each instance *inst* of each function block *FBlock*, we create an automaton $A_{\mathit{FBlock,inst}}$. The locations, transitions and initial location of this automaton are generated using the rules presented below. For each variable in *FBlock* we create a corresponding variable in all the corresponding $A_{\mathit{FBlock,inst}}$ automata. Each automaton contains at least the initial location *init*, the final location *end* and the transition $t_{end}$ from *end* to *init* without any guard.

**Rule PLC4 (Variables)** This rule maps program variables to variables in the IM model.

*Assumption 3.* All variables, except system inputs, that do not have uniquely defined initial values on the PLC platform (e.g., temporary variables, output variables of FCs) are written before they are read. This means that we do not have to model such variables as non-deterministic variables in the IM model, which allows us to limit the state space growth of the generated model.

For each variable $v$ in the program block, there is exactly one corresponding variable $F_V(v)$ in the corresponding automaton. If the variable represents a system input (i.e., variables representing signals coming from the field), it is assigned non-deterministically at the beginning of each PLC cycle.

### B. SCL to IM Transformation

This section presents the rules specific to the SCL to IM transformation.

**Rule SCL1 (SCL statement)** A statement is the smallest standalone element of an SCL program. It can contain other components (e.g., expressions). There are different kinds of statements such as conditional branches, loops and variable

[3]In the case of SCL, we have focused on the representation of the key constructions, and we have omitted the description of e.g., CASE blocks, REPEAT loops, and FOR loops. The handling of expressions, structure and array initializations, and some Siemens-specific constructs (e.g., shared data blocks) are also not discussed, but we have implemented them following the same principles. In the case of SFC, only the action representations are omitted here.

assignments. In this section we define the representation of a single code block consisting of these statements in our IM.

For each statement $stmt$, let $n(stmt)$ be the next statement after $stmt$. Furthermore, for a statement list $sl$, let $\mathit{first}(sl)$ be the first statement of the list. Assumption 1 also applies here. For each SCL statement $stmt$ in the program block, we generate a corresponding location marked as $F_L(stmt)$ in the corresponding automaton. If $stmt$ is the last statement in the program block, $F_L(n(stmt))$ is the location *end* of the corresponding automaton. This general rule is applied to any statement, then more specific rules presented in the following are applied too.

**Rule SCL2 (Variable assignment)** This rule translates SCL variable assignments to IM.

*Assumption 4.* For each "variable access" the variable to be accessed can be determined at transformation time. In particular, this means that pointers are not supported. However, we do support compound variables (arrays and user defined structures). Typically, this is not a restriction as the usage of pointers is not recommended in PLC programs.

For each variable assignment $stmt = \langle v := Expr\rangle$, we add to the corresponding automaton a transition $t = \big(F_L(stmt), TRUE, \langle F_V(v) := Expr\rangle, NONE, F_L(n(stmt))\big)$, going from $F_L(stmt)$ to $F_L(n(stmt))$ with no guard and no synchronization. The assignment associated to the transition updates only the variable $F_V(v)$.

**Rule SCL3 (Conditional statement)** For each conditional statement $stmt = \langle \mathit{IF}\ c\ \mathit{THEN}\ sl_1\ \mathit{ELSE}\ sl_2\ \mathit{END\_IF}\rangle$, we add two transitions to the corresponding automaton:

- $t_1 = (F_L(stmt), c, \langle\,\rangle, NONE, F_L(\mathit{first}(sl_1)))$ goes from $F_L(stmt)$ to $F_L(\mathit{first}(sl_1))$, it has no assignments and no synchronizations, and it has a guard $c$.
- $t_2 = (F_L(stmt), \neg c, \langle\,\rangle, NONE, F_L(\mathit{first}(sl_2)))$ goes from $F_L(stmt)$ to $F_L(\mathit{first}(sl_2))$, it has no assignments, no synchronizations and the guard $\neg c$.

**Rule SCL4 (While loop)** For each while loop $stmt = \langle \mathit{WHILE}\ c\ \mathit{DO}\ sl\ \mathit{END\_WHILE}\rangle$, we add two transitions to the corresponding automaton:

- $t_1 = (F_L(stmt), c, \langle\,\rangle, NONE, F_L(\mathit{first}(sl)))$ goes from $F_L(stmt)$ to $F_L(\mathit{first}(sl))$, it has no assignments and no synchronizations, and it has a guard $c$.
- $t_2 = (F_L(stmt), \neg c, \langle\,\rangle, NONE, F_L(n(stmt)))$ goes from $F_L(stmt)$ to $F_L(n(stmt)))$, it has no assignments, no synchronizations and the guard $\neg c$. This transition corresponds to exiting the loop.

Note that if $stmt$ is a while loop, $n(stmt)$ will denote the next statement *after the loop*. If the last statement of the loop body is $stmt'$, then $n(stmt') = stmt$, as after executing the last statement of the loop body, the next step is to check the condition again.

The for and repeat loops can be expressed based on the rules for conditional branches and while loops.

**Rule SCL5 (FC or FB call)**

*Assumption 5.* All the input variables are assigned in the caller, and all the output variables are assigned in the callee in order to avoid the accessing of uninitialized variables that could contain unpredictable values. Therefore they are not modeled as non-deterministic variables, which allows us to limit the state space growth of the generated model.

For every function (block) call $stmt = \langle [r :=]\mathit{Func}(p_1 := \mathit{Expr}_1, p_2 := \mathit{Expr}_2, \dots)\rangle$ in a code block represented by an automaton $A_{caller}$, we add the following elements. (*Func* can be a function or an instance of a function block, represented by an automaton $A_{callee}$. If *Func* is a function block or a void function, the "$r :=$" part is omitted.)

- A new location $l_{wait}$ is added to $A_{caller}$. It represents the state when the caller block 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 $A_{caller}$, which has no guard and goes from $F_L(stmt)$ to $l_{wait}$. It assigns the function call parameters to the corresponding variables in $A_{callee}$. (It assigns $\mathit{Expr}_1$ to $F_V(p_1)$, etc.)
- A transition $t_2$ is added to $A_{caller}$, which has no guard and goes from $l_{wait}$ to $F_L(n(stmt))$. It assigns *RET_VAL* of the callee to the corresponding variable (variable $F_V(r)$) in $A_{caller}$, if *RET_VAL* exists. It also assigns the corresponding values to the output variables.
- A synchronization $i_1$ is added to the automata network, connecting transition $t_1$ with the first transition of $A_{callee}$.
- A synchronization $i_2$ is added to the automata network, connecting transition $t_{end}$ of $A_{callee}$ with transition $t_2$.

## C. SFC to IM Transformation

This section presents a high-level overview of the rules specific to the SFC to IM transformation.

In the following discussion we do not target the actions that can be assigned to the SFC steps. However, based on the SCL and SFC transformation rules, they can be incorporated easily.

The main idea of the following transformation is that for each SFC step $s$ we create two variables: the *step flag* variable—a variable that indicates if the current step is active, denoted as $s.x$ in the standard and in the Siemens implementation; and another variable that will store a copy of the $s.x$ variables at the beginning of the SFC's call (denoted by $s.x'$ in the following example). The conditions of the transitions will be evaluated on this copy, thus firing of a transition cannot make new transitions enabled.

**Rule SFC1 (SFC step)** For each step ⟨*STEP stepName: END_STEP*⟩ we create a Boolean variable $F_V(\mathit{stepName})$ (representing the variable referenced as *stepName.x* in the PLC programs or in the standard [29]) and a variable $F'_V(\mathit{stepName})$ for internal purposes, both initialized to $FALSE$.

**Rule SFC2 (SFC initial step)** For the initial step ⟨*INITIAL_STEP initStep: END_STEP*⟩, variables are created according to the previous rule. We also add a location $l_0$, and a transition $t_{IM} = (l_0, g, amt, NONE, end)$, where $g = (\neg F_V(\mathit{stepName}_1) \land \neg F_V(\mathit{stepName}_2) \land \dots)$, $amt = \langle F_V(\mathit{initStep}) := TRUE\rangle$. It means that if no steps are active, then the initial step should become active.

Before discussing the representation of transitions we define a set $W = \{w_1, w_2, \dots\}$. Each item of $W$ is a pair $w_i = (S_i, T_i)$ where $S_i$ is a possible "transition input" (step or set of steps occurring in one of the transition's *FROM* part) and $T_i$ is the set of SFC transitions outgoing from $S_i$. The union of $T_i$ sets in $W$ should contain all the transitions of the SFC. A transition input ($S_i$) is typically one single SFC step, but for the transitions closing simultaneous branches it can be composed by multiple SFC steps. These latter transitions have multiple "from" steps and one "to" step. (In the following, let $l_{|W|+1} = \textit{end}$.)

**Rule SFC3 (SFC transitions)**

*Assumption 6.* Based on our experiments with the SFC editor of Siemens we assume that transitions are defined in ascending order of priority in the textual representation[4] that is our input. Also, if there is a transition that is leaving multiple steps at the same time (thus closing a simultaneous branch), there should not be any other transition leaving any of these steps. If this is not respected, the SFC is regarded as syntactically incorrect in the Siemens tools.

For each $w_i = (S_i, T_i) = (\{s_1, s_2, \dots\}, \{t_1, t_2, \dots\})$ in $W$ we create the following IM representation:

We create a location $l_i$. For each $t_j = \langle \textit{TRANSITION tName FROM } s_1, s_2, \dots \textit{ TO } s'_{j1}, s'_{j2}, \dots \textit{ CONDITION} := C \textit{ END\_TRANSITION}\rangle$, we create a transition $t_{IM} = (l_i, g_{IM}, amt, \textit{NONE}, l_{i+1})$. The guard $g_{IM}$ of $t_{IM}$ is a Boolean expression that is only true if (1) $C$ is true, and (2) $F'_V(s_1), F'_V(s_2), \dots$ are true, and (3) all the guards of transitions $t_1, \dots, t_{j-1}$ are false. In other words, (1) the condition of the SFC transition should be satisfied, (2) the input SFC step(s) should be active, and (3) no higher priority event leaving the same SFC step(s) can fire. The assignment is $amt = \langle F_V(s_1) := \textit{FALSE}; F_V(s_2) := \textit{FALSE}; \dots; F_V(s'_{j1}) := \textit{TRUE}; F_V(s'_{j2}) := \textit{TRUE}; \dots\rangle$. (We assume that $t_1, t_2, \dots$ are indexed in descending order of priority.)

Also, for each $w_i$ we add a transition $t'_{IM} = (l_i, g', \langle\rangle, \textit{NONE}, l_{i+1})$ where $g'$ is true iff no other $l_i \to l_{i+1}$ IM transitions are enabled. If no SFC transitions are allowed from $S_i$ (or $S_i$ is not active), this transition allows to proceed to other transitions.

**Rule SFC4 (SFC block)** This rule adds the needed extra information for SFC blocks to the IM.

We add a transition $t_{IM}$ from *init* to $l_0$ that will do a $F'_V(s) := F_V(s)$ assignment for all SFC step $s$. The guard of transition $t_{IM}$ is true iff any of the steps is active.

## D. Reductions on the Intermediate Model

The transformation described above allows us to create an IM representation of a PLC program. However, verification of the models produced from real-life programs is still not feasible with the available tools. In order to address this issue, we apply property-preserving reductions to the IM model. This emphasizes the advantage of using an intermediate model: the reductions are only performed once and propagate automatically to the models generated for the various verification tools.

[4]The first transition in the textual representation has the lowest priority.

- The *Cone of Influence* (COI) reduction eliminates all the variables that do not influence those that contribute to the requirement under analysis.
- *General rule-based reductions* simplify the model by merging states or variables, eliminate unnecessary conditional branches, simplify the Boolean expressions, etc.
- Using the *mode selection*, certain inputs (representing parameters) of the modeled system can be fixed. By introducing these constraints in the IM model instead of in the requirement, the other reduction methods can benefit from this knowledge.

These reductions are presented in more detail in [4].

In addition to the reductions above, we have developed a new method called *variable abstraction*. This technique is an iterative method focused on the verification of simple safety requirements, e.g., "if $\alpha$ is true, then $\beta$ shall be true" ($\mathsf{AG}(\alpha \to \beta)$ in CTL), where $\alpha$ and $\beta$ denote Boolean expressions on variables. This technique builds the abstract models automatically using the variable dependency graph of $\beta$. These models are built by replacing the selected variables with non-deterministic values (similarly to the input variables). Since these variables do not depend on any others, the COI algorithm can eliminate more variables from the model.

Fig. 2 shows a simple variable dependency graph for the requirement $\mathsf{AG}(a \to b)$ (so $\alpha = \{a\}$, $\beta = \{b\}$ in this example). In this graph, nodes represent variables. The gray variables are part of the requirement ($a$ and $b$) and the edges represent dependencies (e.g., an assignment or a conditional statement). We defined a distance metric $\delta$ for each variable of the graph. Its value is the smallest distance from a variable in $\beta$. In the $i$th iteration, the variables with $\delta = i$ are replaced by non-deterministic values and the variables with $\delta > i$ are deleted. If for any $a \in \alpha$, $\delta(a) > i$, then $a$ is replaced by a non-deterministic value instead of deleting it.

In the first iteration of the example the variables to be replaced by non-deterministic values are: $a$, $y$ and $x$. If the verification result is true, then the safety requirement is satisfied on the original model, as the abstract model is an over-approximation of the original one. If the verification result is false and it cannot be determined if the counterexample is real or spurious, a new iteration is needed.

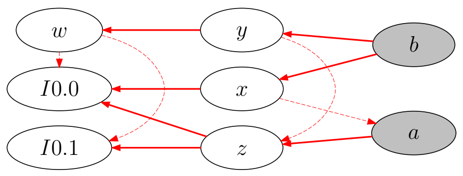


Figure 2. Variable dependency graph of an example PLC program

More precisely, in order to abstract a set of variables $V$, we perform the following steps on the Abstract Syntax Tree of the PLC code, for each $v \in V$:

- All assignments of the variable $v$ are removed.
- An assignment $v := \textit{undefined}$ is added at the beginning of the scan cycle, meaning that the value of $v$ will be undefined and it will take any value from its domain.

This technique is sound, i.e., if a safety property holds after variable abstraction, it holds in the original system. However, it is not complete, meaning that spurious safety violations can be detected, since variable abstraction generates behaviors not present in the original system. Such spurious violations can be detected by analyzing the counterexamples. Variable abstraction is illustrated in Section V and its implications on the verification process are discussed in Section VI.

### *E. IM–nuXmv Transformation*

The IM model representing the PLC code has to be transformed into the concrete syntax of one or more model checking tools to verify the given requirements. Our methodology is general and can be applied for any model checker with an input language based on automata or transition systems. Here, we briefly introduce the transformation from IM to the input language of nuXmv [32] as an example.

For each automaton $A$ in the IM model, we create a module $M_A$ in the nuXmv model with exactly one instance. Each variable in $A$ is represented by a variable in the module $M_A$. Furthermore, a variable *loc* is added to each module $M_A$ that represents the current location of the automaton $A$.

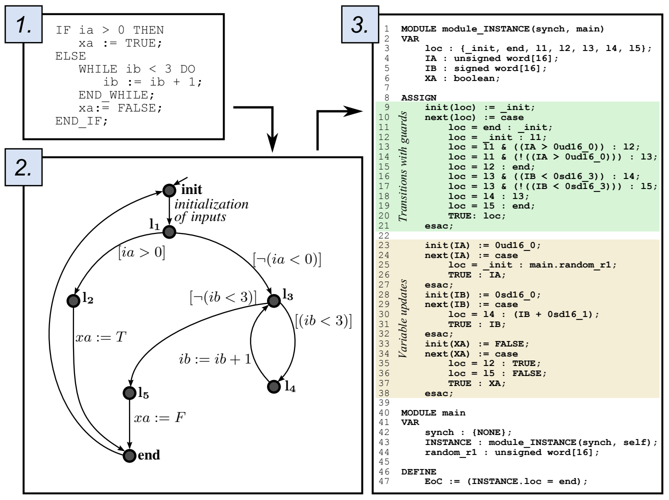
A module *main* is also created in the nuXmv model containing a variable *synch* enumerating all possible synchronizations and the value *NONE*. At each cycle, this variable encodes the synchronization to be performed.

### *F. Transformation Examples*

The following shows two examples of the transformations described in this section.

Fig. 3 shows an example transformation from SCL code (1) through IM model (2) to nuXmv model (3). The SCL code contains a conditional statement, a while loop and three variable assignments to Boolean and integer variables. In Part 2 of Fig. 3, one can observe the true and false branches of the conditional statement ($l_1 - l_2 - \dots$ and $l_1 - l_3 - \dots$) and the representation of the while loop ($l_3 - l_4$). The key ideas of the transformation to nuXmv can be seen in the Part 3. The variable `loc` defined in line 3 represents the locations of the automaton. The transitions and guards are defined by the case statement in lines 9–21 (e.g., line 13 represents the transition $l_1 \to l_2$ with guard $[ia > 0]$). The variable updates are given separately in lines 23–38 (e.g., "`loc = l4 : (IB + 0sd16_1)`" in line 30 for the variable update $ib := ib + 1$ of transition $l_4 \to l_5$). The global structure (e.g., main module, instances, random value handling) of the generated nuXmv model can also be observed.

Part 1 of Fig. 4 shows an example SFC program[5]. The steps are represented by gray boxes (S1, S2, etc.), the transitions are represented by black rectangles. The transition T1 opens a simultaneous branch (denoted by double line), thus after firing T1, S2 and S3 will be both active. On the contrary, S3 is followed by an alternative branch: either T2 or T3 can fire. If T2 fires, S4 will be active, if T3 fires, S5 can be active. T6 will only fire if S2 and S6 are both active.



```
IF ia > 0 THEN
    xa := TRUE;
ELSE
    WHILE ib < 3 DO
        ib := ib + 1;
    END_WHILE;
    xa:= FALSE;
END_IF;
```

```
1   MODULE module_INSTANCE(synch, main)
2   VAR
3       loc : {_init, end, l1, l2, l3, l4, l5};
4       IA : unsigned word[16];
5       IB : signed word[16];
6       XA : boolean;
7
8   ASSIGN
9       init(loc) := _init;
10      next(loc) := case
11          loc = end : _init;
12          loc = _init : l1;
13          loc = l1 & ((IA > 0ud16_0)) : l2;
14          loc = l1 & (!((IA > 0ud16_0))) : l3;
15          loc = l2 : end;
16          loc = l3 & ((IB < 0sd16_3)) : l4;
17          loc = l3 & (!((IB < 0sd16_3))) : l5;
18          loc = l4 : l3;
19          loc = l5 : end;
20          TRUE: loc;
21      esac;
22
23      init(IA) := 0ud16_0;
24      next(IA) := case
25          loc = _init : main.random_r1;
26          TRUE : IA;
27      esac;
28      init(IB) := 0sd16_0;
29      next(IB) := case
30          loc = l4 : (IB + 0sd16_1);
31          TRUE : IB;
32      esac;
33      init(XA) := FALSE;
34      next(XA) := case
35          loc = l2 : TRUE;
36          loc = l5 : FALSE;
37          TRUE : XA;
38      esac;
39
40  MODULE main
41  VAR
42      synch : {NONE};
43      INSTANCE : module_INSTANCE(synch, self);
44      random_r1 : unsigned word[16];
45
46  DEFINE
47      EoC := (INSTANCE.loc = end);
```

Figure 3. Example SCL–IM–nuXmv translation

Each transition can only fire if its condition is evaluated to true. The conditions are not shown on the figure but for each T$i$ the corresponding condition (guard) is the Boolean variable $C_i$. The corresponding IM model is shown in Part 2 of Fig. 4. For each step S$i$ the corresponding $F_V(\mathrm{S}i)$ variable is denoted by s$i.x$, the corresponding $F'_V(\mathrm{S}i)$ variable is denoted by s$i.x'$. The reason for using both $x$ and $x'$ is to avoid *transition chaining*, i.e., when the firing of the transition enables another transition that fires too. For example, firing T6 can enable T7 (providing that $C_7$ is true), but according to the semantics of Siemens SFCs this firing can only performed when the SFC is called for the next time. The parallel activation of S2 and S3 can be observed in the variable updates of the $l_1 \to l_2$ IM transition. The alternative activation of S4 and S5 is visible in the $l_2 \to l_3$ IM transitions.

The set $W$ defined for the transformation (cf. Rule SFC3) is the following in this example: $W = \{(\{\mathrm{S1}\}, \{\mathrm{T1}\}), (\{\mathrm{S2}, \mathrm{S6}\}, \{\mathrm{T6}\}), (\{\mathrm{S3}\}, \{\mathrm{T2}, \mathrm{T3}\}), (\{\mathrm{S4}\}, \{\mathrm{T4}\}), (\{\mathrm{S5}\}, \{\mathrm{T5}\}), (\{\mathrm{S7}\}, \{\mathrm{T7}\})\}$.

The first and the last transition of the IM model contain the synchronizations with other automata. These synchronizations represent the FB call of the SFC block.

### *G. Implementation of the Methodology*

The methods presented above are implemented in a proof-of-concept tool called *PLCverif*. The PLC input parser is implemented using Xtext[6]. The provided abstract syntax tree is the input of the transformation and reduction algorithms implemented in Java. The whole procedure is implemented in an Eclipse-based tool that allows the user to import the PLC code and define the requirement to be verified. It also performs the model transformations, the automated model reductions and calls the model checker tools. The feedback provided to the user is a verification report containing the result of the verification and the eventual counterexample. The definition of the code and requirement to be verified is the only task of the user, all the rest is automated and hidden.

[5]Note that it is a directed graph, but in Siemens notation the arrows are only shown if direction is not top to bottom.

[6]http://eclipse.org/Xtext/

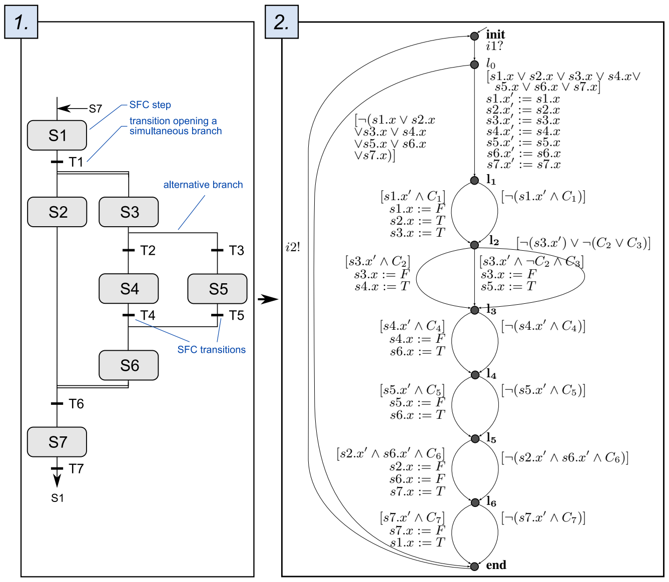

1.
S7
SFC step
transition opening a simultaneous branch
S1
T1
alternative branch
S2
S3
T2
T3
S4
S5
T4
T5
SFC transitions
S6
T6
S7
T7
S1
2.
init
i1?
l0
[s1.x ∨ s2.x ∨ s3.x ∨ s4.x∨ s5.x ∨ s6.x ∨ s7.x]
s1.x' := s1.x
s2.x' := s2.x
s3.x' := s3.x
s4.x' := s4.x
s5.x' := s5.x
s6.x' := s6.x
s7.x' := s7.x
[¬(s1.x ∨ s2.x ∨s3.x ∨ s4.x ∨s5.x ∨ s6.x ∨s7.x)]
l1
[s1.x' ∧ C1] s1.x := F s2.x := T s3.x := T
[¬(s1.x' ∧ C1)]
l2
[¬(s3.x') ∨ ¬(C2 ∨ C3)]
i2!
[s3.x' ∧ C2] s3.x := F s4.x := T
[s3.x' ∧ ¬C2 ∧ C3] s3.x := F s5.x := T
l3
[s4.x' ∧ C4] s4.x := F s6.x := T
[¬(s4.x' ∧ C4)]
l4
[s5.x' ∧ C5] s5.x := F s6.x := T
[¬(s5.x' ∧ C5)]
l5
[s2.x' ∧ s6.x' ∧ C6] s2.x := F s6.x := F s7.x := T
[¬(s2.x' ∧ s6.x' ∧ C6)]
l6
[s7.x' ∧ C7] s7.x := F s1.x := T
[¬(s7.x' ∧ C7)]
end

Figure 4. Example SFC–IM translation

PLCverif is not available yet, but our plan is to make it production-ready and downloadable from our website[7].

## V. Experimental Results

This section provides experimental verification results on CERN PLC programs and real requirements. Most of the control systems used at CERN are developed using the UNICOS (Unified Industrial Control System) framework. This framework provides a common development process and a library of reusable base objects representing frequently-used industrial control instrumentation (e.g., sensors, actuators). We present two typical formal verification use cases. In the first example, we check requirements on the model of a single base object from the UNICOS library consisting mainly of a single function block with some function calls. The second example shows the verification of a requirement on a complete UNICOS application controlling a cryogenics subsystem. This application consists of hundreds of base object instances and a large application-specific logic.

### A. Verification of a UNICOS Base Object

In this section, we present the verification of the UNICOS base object `OnOff`, which represents an actuator driven by digital signals (e.g., valve, heater, motor). This object can run in different configurations with different parameters and in various modes; it can handle various errors.

In the PLC code, the `OnOff` object is implemented by a function block written in SCL. This function block has 600 lines of code, 60 input variables and 62 output variables. The data types used in this block are Booleans, integers, arrays, floats and structures, e.g., an array composed of these data types. The function block has several function calls to 3 different functions.

[7]http://cern.ch/plcverif/

Table I
Metrics of the models of OnOff

| Metric | Non-reduced model | Reduced model |
|---|---|---|
| PSS | $1.6 \cdot 10^{218}$ | $4.3 \cdot 10^{26}$ |
| Model generation | $0.2\ s$ | $0.6\ s$ |
| Verification | — | $19.4\ s$ |
| | | ($6.1\ s$ without c.ex.) |

The following is a real requirement expressed informally by the UNICOS developers: *if the object is controlled locally (is in the so-called "Hardware" mode) and there is no interlock, nor explicit output change request valid in this mode, the output keeps its value.*

To help developers to express real requirements and to facilitate the cooperation between developers and formal verification experts, we defined a set of easy-to-use requirement patterns (for details see [33]). Using this patterns, a developer was able to formalize the requirement using variables and Boolean expressions as follows: "If *OutOnOV=false & TStopI=false & FuStopI=false & StartI=false* is true at the end of cycle $N$ and *HLD=true & HOnR=false & HOffR=false & TStopI=false & FuStopI=false & StartI=false* is true at the end of cycle $N+1$, then *OutOnOV=false* is always true at the end of cycle $N+1$." Requirements expressed using our patterns can be automatically formalized in LTL:

$$\mathsf{G}\Big(\big(EoC \wedge \neg OutOnOV \wedge \neg TStopI \wedge \neg FuStopI \wedge \neg StartI \wedge HLD \wedge \mathsf{X}(\neg EoC\,\mathsf{U}(EoC \wedge HLD \wedge \neg HOnR \wedge \neg HOffR \wedge \neg TStopI \wedge \neg FuStopI \wedge \neg StartI))\big) \rightarrow \mathsf{X}(\neg EoC\,\mathsf{U}(EoC \wedge \neg OutOnOV))\Big)$$

In this formula, $EoC$ is a Boolean symbol, which evaluates to true at the end of each PLC cycle and only then.

Table I summarizes the performance metrics of the approach[8]. Before the reductions, the size of the potential state space (PSS) is $1.6 \cdot 10^{218}$. After the general and requirement-dependent reductions the PSS has $4.3 \cdot 10^{26}$ states, whereof $4.9 \cdot 10^{14}$ are reachable. Evaluation of the requirement takes 6.1 s using nuXmv without counterexample generation. This showed that the requirement is not satisfied. If the counterexample is generated too, the run time is 19.4 s. The generation of the model including all the reductions takes 0.6 s.

We used the counterexample generated by nuXmv to prove that the bug detected in our model is, indeed, a real bug. To this end, we have analyzed the counterexample and automatically generated a PLC program, exhibiting the bug on real hardware using the real code of the base object. This generated PLC code drives the module under verification to a state where the requirement is violated by feeding it with appropriate input values extracted from the counterexample. Thus, the discovered bug was not a result of our model generation technique, but was also confirmed in the real PLC code.

This methodology has been found very useful for the controls engineer. We have verified 52 different requirements

[8]The measurements were performed on a PC with Intel® Core™ i7-3770 3.4 GHz CPU, 8 GB RAM, on Windows 7 x64.

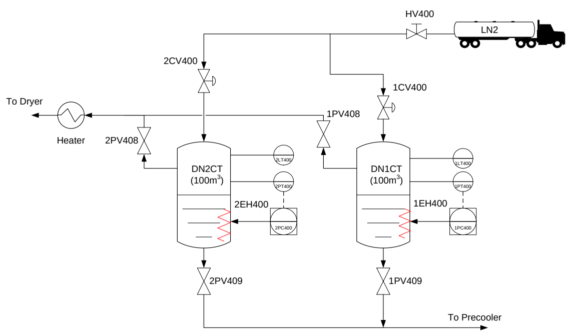


Figure 5. QSDN process

provided by UNICOS developers for the OnOff object. Our experiments identified 11 cases when the requirement was not satisfied on this well-tested object used in numerous CERN applications. In 4 cases, the PLC program had to be modified. In 7 cases, the problem was due to incomplete or bad specification.

## B. Verification of a Full UNICOS PLC Application

We have chosen the so-called QSDN application[9] as a second case study. This application controls a subsystem of the Large Hadron Collider cryogenics control system. The size of QSDN is representative of medium size UNICOS applications. It contains 110 functions and function blocks, and consists of approximately 17,500 lines of code. Before reductions, this results in a huge generated model: the IM contains 302 automata, and the PSS size is $10^{31985}$ (see $M_1$ in Table II).

Verification of a full UNICOS application may rely on specifications of base objects of the UNICOS library, instead of their implementation. The correctness of these objects is addressed separately (cf. Section V-A). Thus, we speed-up the verification process and focus the analysis on the potential integration errors without compromising its soundness. This is in contrast with testing, where a faulty base object could potentially hide integration errors.

Thus the goal is to check the application-specific logic implemented in SCL and SFC. This logic is described in the *UNICOS functional analysis* document, which is a semiformal textual specification. Application-specific functional requirements are also extracted from this specification.

Table II presents metrics relevant to the generated models. The original state space is huge and the original model obviously cannot be verified (see $M_1$ in Table II). Based on the requirement to be verified, both the general and requirement-specific reductions can be used to reduce the model. Although these techniques have shown their efficiency and lead to a considerable state space reduction, the reduced model is still huge and impossible to verify.

The requirements extracted from the functional analysis are typically simple safety requirements, e.g., "if $\alpha$ is true, then $\beta$ is true" (in CTL, $\mathsf{AG}(\alpha \rightarrow \beta)$), where $\alpha$ and $\beta$ denote Boolean expressions on variables.

[9]QSDN stands for Cryogenics Surface Liquid Nitrogen Storage System.

```
(* Position Management*)
QSDN_4_1EH4001Ok.AuOnR :=
   QSDN_4_DN1CT_SEQ_DB.OkSignalOn.x OR
   QSDN_4_DN1CT_SEQ_DB.ValvesOn.x OR
   QSDN_4_DN1CT_SEQ_DB.Run.x;

QSDN_4_1EH4001Ok.AuOffR :=
   NOT QSDN_4_1EH4001Ok.AuOnR;
```

Figure 6. Excerpt of QSDN code relevant to the case study

Table II
METRICS OF THE MODELS OF QSDN

| Metric | Non-reduced model ($M_1$) | Reduced model ($M_2$) | Abstract, reduced model ($M_3$) |
|---|---|---|---|
| PSS | $10^{31985}$ | $10^{5048}$ | $2 \cdot 10^{48}$ (RSS: $3 \cdot 10^{42}$) |
| Variables | 31,402 | 3757 | 32 |
| Model generation | 4.2 $s$ | 15.3 $s$ | 50.2 $s$ |
| Verification | — | — | 4.5 $s$ |

The example requirement to be checked is the following: "If `QSDN_4_DN1CT_SEQ_DB.Stop.x` is true (at the end of a scan cycle), `QSDN_4_1EH4001Ok.AuOffR` should be true also." Fig. 6 shows the relevant part of the QSDN PLC code. Satisfaction of the requirement cannot be shown by inspection of this code part and requires additional information from the rest of the application. After the COI reduction 3757 variables are kept in the reduced model (see model $M_2$ in Table II). Formal verification is still not possible.

In this example, 4 iterations of the variables abstraction were needed to prove that the requirement is satisfied on the formal model. Using this technique, less than a minute was necessary to check this requirement (see $M_3$ in Table II). As can be seen, our verification method can be used for isolated verification of modules or for verification of complete PLC applications. Approximately 30 different requirements were extracted from the QSDN functional analysis document, all of which were proven to be satisfied using the above method.

## VI. Analysis and Discussion

The proposed methodology allows the generation and analysis of formal models for real-life systems. Using these techniques, we have identified bugs in real-life systems deployed at CERN.

Verification is made possible by the reduction techniques applied to the Intermediate Model (IM) representation of such systems. Among the techniques discussed in the paper, COI, general rule-based reductions and mode selection preserve the meaning of the model as relevant to a specific property: the property is satisfied in the reduced model if and only if it is satisfied in the original one.

The variable abstraction technique adds spurious system behaviors by introducing non-determinism. If a safety property holds in the reduced model, it holds in the original one. However, spurious counterexamples may occur. If the property does not hold in the abstract model and a counterexample is produced, further analysis must be performed to determine whether it represents a real bug. Such analysis requires the expertise of a developer and knowledge of the application.

If the counterexample represents a possible behavior of the system, a real bug is identified and the verification process terminates. If this analysis cannot prove that the counterexample is spurious, the model needs to be refined to reduce abstraction. In particular, application developers can refine a model by providing invariants, i.e., properties that are known to be satisfied in all reachable states of the model. For example, the statement "Two steps of an SFC program cannot be both active at the end of the same PLC cycle" is an invariant satisfied by all SFC programs that do not contain parallel branches. The above process is applied iteratively until the requirement is shown to be satisfied or a bug is identified.

In the current state of the methodology, we cannot prove mathematically the correctness of all our model transformations. However, when a discrepancy between the specification and the formal model is detected, we can prove that this bug exists in the real PLC program. A small piece of code called "PLC demonstrator" can be automatically generated out of the counterexample give by the model checker. This code reproduces the combination of input variable values that provoke the discrepancy and the monitor will check if the bug is reproduced also in the real PLC program.

### *Correctness of Our Approach*

Apart from the combinatorial explosion of the state space, a fundamental limitation of all verification methodologies lies in the fact that requirements to be satisfied by the system are usually expressed informally. Moreover, they may be inconsistent [34] or fail to reflect the precise behavior expected by the designers [35]. To address this problem, we have defined a set of easy-to-use requirement patterns [33]. Our approach is similar to the one widely adopted in the industry, where simplified formal languages are used by developers to define requirements [36]–[38].

The second limitation comes from the correctness of the model checking tools and of the result interpretation by the developers. Although several model checkers are used in a variety of projects, none of the well-established model checkers have been formally verified themselves. Furthermore, most practical verification methodologies involve abstraction, leading to the possibility of spurious counterexamples. As with compilation warnings, developers tend to dismiss counterexamples as spurious, whenever they cannot be easily confirmed. Although the latter problem can be partially addressed by making counterexample analysis part of the automated process (e.g., [18]), the former is likely to persist. Thus, model checking cannot be the sole basis for system certification. However, we have shown that many bugs can be identified by formal verification, which escape the usual testing procedures, considerably increasing the confidence one can put into industrial control applications.

A mathematically sound correctness proof requires the establishment of formal semantics for all involved languages. Using operational semantics for the model checker languages, our intermediate representation (cf. [3]) and the languages of the IEC 61131-3 standard (cf. [39] for IL and FBD), we can establish a simulation relation between the original and transformed models: states in the original model are related with semantically equivalent states in the transformed model. The simulation relation must preserve the properties to be verified (the choice of the employed abstractions influences the classes of properties that are preserved [27]). One such complete proof of the correctness of abstraction rules that we used can be found in [5].

## VII. Conclusion and Future Work

We have presented a general automated methodology for formal verification of PLC programs. The methodology is based on an intermediate model (IM), used as a pivot between all the PLC and formal modeling languages that we use. This approach potentially covers all PLC languages. Current implementation supports SCL and SFC; the support for IL is under development. The IM model is automatically reduced, following which models for different verification tools are automatically generated. This allows us to benefit from the combined strengths of the different verification tools. Current implementation allows the generation of nuXmv, UPPAAL and BIP models. We have presented the most relevant transformation rules from SCL and SFC to IM. The reduction and abstraction techniques presented in the paper are applied to the intermediate model. On one hand this makes them independent from the source language, allowing verification of heterogeneous PLC applications. On the other hand, this approach decouples these techniques from the choice of the model checker, allowing greater flexibility and coherence of verification results. Finally, we have applied the presented methodology to real-life PLC control systems developed at CERN, demonstrating the feasibility of our approach. Formal verification using the presented methodology has allowed us to identify bugs in these systems, which have escaped the standard testing procedures.

There are two main directions for the future work on this project. First is the improvement of the specification methods for control systems. Such a language must be formal, unambiguous and easy-to-understand. Second is the improvement of the abstraction techniques with the goal of automatizing the variable abstraction.

## References

[1] *IEC 61508: Functional Safety of Electrical / Electronic / Programmable Electronic Safety-related Systems*, IEC Std., 2010.

[2] B. Fernández Adiego, E. Blanco Viñuela, J.-C. Tournier, V. M. González Suárez, and S. Bliudze, "Model-based automated testing of critical PLC programs," in *Proc. of the 11th IEEE Int. Conf. on Industrial Informatics*, 2013, pp. 722–727.

[3] B. Fernández Adiego, D. Darvas, J.-C. Tournier, E. Blanco Viñuela, J. O. Blech, and V. M. González Suárez, "Automated generation of formal models from ST control programs for verification purposes," CERN, Internal Note CERN-ACC-NOTE-2014-0037, 2014. [Online]. Available: http://cds.cern.ch/record/1708853/files/Internal%20Note.pdf

[4] D. Darvas, B. Fernández Adiego, A. Vörös, T. Bartha, E. Blanco Viñuela, and V. M. González Suárez, "Formal verification of complex properties on PLC programs," in *Formal Techniques for Distributed Objects, Components, and Systems*, ser. LNCS. Springer, 2014, vol. 8461, pp. 284–299.

[5] 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 the 19th IFAC World Congress*. IFAC, 2014, pp. 3333–3339.

[6] G. Frey and L. Litz, "Formal methods in PLC programming," in *IEEE Int. Conf. on Systems, Man, and Cybernetics*, vol. 4. IEEE, 2000, pp. 2431–2436.

[7] A. Mader and H. Wupper, "Timed automaton models for simple programmable logic controllers," in *Proc. of 11th Euromicro Conference on Real-Time Systems*. IEEE, 1999, pp. 106–113.

[8] A. Sülflow and R. Drechsler, "Verification of PLC programs using formal proof techniques," in *Formal Methods for Automation and Safety in Railway and Automotive Systems*. L'Harmattan, 2008, pp. 43–50.

[9] D. Soliman, K. Thramboulidis, and G. Frey, "Transformation of function block diagrams to UPPAAL timed automata for the verification of safety applications," *Annual Reviews in Control*, vol. 36, no. 2, pp. 338–345, 2012.

[10] M. Perin and J.-M. Faure, "Building meaningful timed models of closed-loop DES for verification purposes," *Control Engineering Practice*, vol. 21, no. 11, pp. 1620–1639, 2013.

[11] H. B. Mokadem, B. Bérard, V. Gourcuff, O. De Smet, and J.-M. Roussel, "Verification of a timed multitask system with UPPAAL," *IEEE Transactions on Automation Science and Engineering*, vol. 7, no. 4, pp. 921–932, 2010.

[12] C. A. Sarmento, J. R. Silva, P. E. Miyagi, and D. J. Santos Filho, "Modeling of programs and its verification for programmable logic controllers," in *Proc. of the 17th IFAC World Congress*. IFAC, 2008, pp. 10 546–10 551.

[13] O. Pavlović and H.-D. Ehrich, "Model checking PLC software written in function block diagram," in *Proc. of the Int. Conf. on Software Testing, Verification and Validation*. IEEE, 2010, pp. 439–448.

[14] D. Soliman and G. Frey, "Verification and validation of safety applications based on PLCopen safety function blocks," *Control Engineering Practice*, vol. 19, pp. 929–946, 2011.

[15] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen, "Towards the automatic verification of PLC programs written in Instruction List," in *Proc. of the IEEE Int. Conf. on Systems, Man, and Cybernetics*, vol. 4. IEEE, 2000, pp. 2449–2454.

[16] T. Bartha, A. Vörös, A. Jámbor, and D. Darvas, "Verification of an industrial safety function using coloured Petri nets and model checking," in *Proc. of the 14th Int. Conf. on MITIP 2012*. Computer and Automation Research Inst., 2012, pp. 472–485.

[17] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg, "Verification of PLC programs given as sequential function charts," in *Integration of Software Specification Techniques for Applications in Engineering*, ser. LNCS. Springer, 2004, vol. 3147, pp. 517–540.

[18] S. Biallas, J. Brauer, and S. Kowalewski, "Counterexample-guided abstraction refinement for PLCs," in *Proc. of the 5th Int. Conf. on Systems Software Verification*. USENIX Association, 2010.

[19] J. Yoo, S. Cha, and E. Jee, "A verification framework for FBD based software in nuclear power plants," in *Proc. of 15th Asia-Pacific Software Engineering Conference*. IEEE, 2008, pp. 385–392.

[20] R. Glück and F. Krebs, "Towards interactive verification of programmable logic controllers using modal Kleene algebra and KIV," in *Relational and Algebraic Methods in Computer Science*, ser. LNCS. Springer, 2015, vol. 9348, pp. 241–256.

[21] J. Sadolewski, "Conversion of ST control programs to ANSI C for verification purposes," *e-Informatica*, vol. 5, no. 1, pp. 65–76, 2011.

[22] V. Gourcuff, O. de Smet, and J.-M. Faure, "Improving large-sized PLC programs verification using abstractions," in *Proc. of the 17th IFAC World Congress*. IFAC, 2008, pp. 5101–5106.

[23] T. Lange, M. R. Neuhäußer, and T. Noll, "Speeding up the safety verification of programmable logic controller code," in *Hardware and Software: Verification and Testing*, ser. LNCS. Springer, 2013, vol. 8244, pp. 44–60.

[24] J. Nellen, E. Ábrahám, and B. Wolters, "A CEGAR tool for the reachability analysis of PLC-controlled plants using hybrid automata," in *Formalisms for Reuse and Systems Integration*, ser. Advances in Intelligent Systems and Computing. Springer, 2015, vol. 346, pp. 55–78.

[25] E. Kuzmin and V. Sokolov, "Modeling, specification and construction of PLC-programs," *Automatic Control and Computer Sciences*, vol. 48, no. 7, pp. 554–563, 2014.

[26] T. Ovatman, A. Aral, D. Polat, and A. O. Ünver, "An overview of model checking practices on verification of PLC software," *Software & Systems Modeling*, 2014, advance online publication, DOI: 10.1007/s10270-014-0448-7.

[27] M. Bozga, S. Graf, L. Mounier, and I. Ober, "Modeling and verification of real time systems using the IF toolbox," in *Real Time Systems 1: Modeling and verification techniques*, ser. Traité IC2, série Informatique et systèmes d'information. Hermes, Lavoisier, 2008, vol. 1, ch. 9.

[28] ——, "IF validation environment tutorial," in *Model Checking Software*, ser. LNCS, 2004, vol. 2989, pp. 306–307.

[29] *IEC 61131: Programming languages for programmable logic controllers*, IEC Std., 2013.

[30] Siemens, *SIMATIC Programming with STEP7 – Manual*, 2010, A5E02789666-01.

[31] ——, "Standards compliance according to IEC 61131-3," 2011. [Online]. Available: http://support.automation.siemens.com/WW/view/en/50204938

[32] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker," in *Computer Aided Verification*, ser. LNCS. Springer, 2014, vol. 8559, pp. 334–342.

[33] B. Fernández Adiego, D. Darvas, J.-C. Tournier, E. Blanco Viñuela, and V. M. González Suárez, "Bringing automated model checking to PLC program development – A CERN case study," in *Proc. of the 12th Int. Workshop on Discrete Event Systems*. IFAC, 2014, pp. 394–399.

[34] R. Alur, D. Arney, E. L. Gunter, I. Lee, J. Lee, W. Nam, F. Pearce, S. Van Albert, and J. Zhou, "Formal specifications and analysis of the computer-assisted resuscitation algorithm (CARA) infusion pump control system," *International Journal on Software Tools for Technology Transfer*, vol. 5, no. 4, pp. 308–319, 2004.

[35] G. Klein, K. Elphinstone, G. Heiser *et al.*, "seL4: Formal verification of an OS kernel," in *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 2009, pp. 207–220.

[36] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in property specifications for finite-state verification," in *Proc. of the 21st Int. Conf. on Software Engineering*. ACM, 1999, pp. 411–420.

[37] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh, "The temporal logic Sugar," in *Computer Aided Verification*, ser. LNCS. Springer, 2001, vol. 2102, pp. 363–367.

[38] R. Armoni, L. Fix, A. Flaisher *et al.*, "The ForSpec temporal logic: A new temporal property-specification language," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2002, vol. 2280, pp. 296–311.

[39] J. O. Blech and S. O. Biha, "Verification of PLC properties based on formal semantics in Coq," in *Software Engineering and Formal Methods*, ser. LNCS. Springer, 2011, vol. 7041, pp. 58–73.