Formal Verification of Complex Properties on PLC Programs

Dániel Darvas, Borja Fernández Adiego, András Vörös, Tamás Bartha, Enrique Blanco Viñuela, Víctor M. González Suárez

Formal verification of complex properties on PLC programs

Dániel Darvas[1], Borja Fernández Adiego[1], András Vörös[2], Tamás Bartha[3], Enrique Blanco Viñuela[1], and Víctor M. González Suárez[4]

[1] CERN, European Organization for Nuclear Research, Geneva, Switzerland, {bfernand,ddarvas,eblanco}@cern.ch,
[2] Budapest University of Technology and Economics, Budapest, Hungary, vori@mit.bme.hu
[3] Institute for Computer Science and Control, Hungarian Academy of Sciences, Budapest, Hungary, bartha.tamas@sztaki.mta.hu
[4] University of Oviedo, Gijón, Spain, victor@isa.uniovi.es

Abstract. Formal verification has become a recommended practice in the safety-critical application areas. However, due to the complexity of practical control and safety systems, the state space explosion often prevents the use of formal analysis. In this paper we extend our former verification methodology with effective property preserving reduction techniques. For this purpose we developed general rule-based reductions and a customized version of the Cone of Influence (COI) reduction. Using these methods, the verification of complex requirements formalised with temporal logics (e.g. CTL, LTL) can be orders of magnitude faster. We use the NuSMV model checker on a real-life PLC program from CERN to demonstrate the performance of our reduction techniques.

Keywords: PLC, model checking, automata, temporal logic, reduction, cone of influence, NuSMV

1 Introduction

At CERN (European Organization for Nuclear Research) large industrial installations, such as cryogenics, HVAC (heating, ventilation, and air conditioning) and vacuum systems rely on PLC (Programmable Logic Controller) based control systems. These systems are critical to CERN's operation and guaranteeing that their behaviours conform to their requirement specifications is of highest importance. Formal verification, and especially model checking is a promising technique to ensure that PLC programs meet their specifications.

The difficulties of bringing model checking techniques to the automation industry are twofold. First, both a formal model of the system to be verified and a formal specification of the verification criteria need to be created. They are usually the result of a cooperation between a control engineer and a formal methods expert, with potential for misunderstanding. A second concern is the amount of necessary computation resources. Therefore, the formal models for

verifying industrial control systems should be generated automatically, making the verification and validation process less complex and error-prone for the control engineers. The general models created automatically are usually huge for real applications, thus this generation should be extended with reduction techniques.

Our goal is to verify complex requirements coming from the real design and development process. To formalise those requirements, we need to use powerful temporal logic, as the properties to be checked cannot be expressed by simple Boolean logic. Here we target Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) expressions. We proposed previously a methodology for automated modelling and verification of PLC programs [1]. In this paper we focus on reduction techniques making the automated verification of complex properties quicker on real PLC programs. We present general rule-based reductions and a customized version of the cone of influence (COI) reduction tailored to our intermediate models. Experimental verification results on a CERN's real-life ST (Structured Text) [2] program using the NuSMV model checker are also included in the paper.

The rest of the paper is structured as follows: the next part overviews the work related to abstraction techniques applied for PLC verification. Section 2 introduces the general methodology designed to automatically generate formal models out of PLC programs. Section 3 presents the property preserving reduction techniques we apply. Section 4 analyses the proposed techniques showing verification results on a real case study. Finally, Section 5 concludes the paper.

Related Work Although abstraction and reduction techniques for software verification have previously been studied by other authors, none of them provides a complete workflow to generate and verify PLC programs automatically, especially for programs written in ST language (the most common PLC development language in CERN). In this section, the existing results about reduction techniques applied to PLC systems are discussed.

There were multiple attempts to verify PLC programs using model checking [3–7], but these do not include any specific technique to handle the state space explosion problem automatically, therefore they cannot be applied for our purposes.

In [8] the authors address the problem of state explosion of formal models derived from IL (Instruction List) PLC programs by using an algorithm based on data and interpretation abstraction. Their algorithm has several limitations, e.g. only Boolean variables can be used. In [9], bounded model checking is applied to PLC programs. A simple intermediate model is used to translate PLC programs written in IL language to formal models. Reduction techniques such as constant folding, slicing, and forward expression propagation are employed to optimize the models for bounded model checking. However, this approach can only check reachability, so it does not support the more general properties necessary for our complex requirements. Similarly in [10], a method for verifying PLC programs written in IL using the Counterexample-Guided Abstraction Refinement (CEGAR) is presented. The limitations of the approach are the same as for other CEGAR approaches: it can handle only safety properties.

As can be seen from the extensive research in this field, the verification of PLC systems is an important task, and our work extends the former results in order to be suitable for real industrial applications and complex properties.

2 Motivation

This section describes the complexity issues of verifying PLC programs. First, we introduce the application environment and the control systems at CERN, then we outline the modelling methodology.

2.1 Application Environment

PLCs are embedded computers applied for industrial control and monitoring tasks. This application domain requires quick and reliable software. Therefore the PLC programs have simple structure and static memory handling, but large number of I/O variables. To have consistent inputs and outputs, they perform a cyclic process called *scan cycle* consisting of three subsequent steps: (1) reading the actual values to the memory, (2) interpreting and executing the implemented PLC program, and (3) assigning the computed values to the real output peripheries. PLCs can be programmed using several different languages [2]. Here we focus ST (Structured Text) language, as it is widely used in CERN. ST is a syntactically Pascal-like, high level language to describe function blocks of controllers (see Fig. 1 for a small ST example). The detailed introduction of the language can be read in [2]. The building elements of ST are functions and function blocks ("stateful functions"), called from the cyclic main program.

At CERN, control systems are developed by using the UNICOS (Unified Industrial Control System) framework [11]. It provides a library of base objects representing common industrial control instrumentation (e.g. sensor, actuators, subsystems). These objects are expressed as function blocks in ST language. The correctness of the UNICOS library and the PLC programs are crucial, as a failure can cause significant damage in the equipment, as well as delay in the experiments. Our main goal is to apply model checking to these baseline objects. The PLC programs have finite set of states and finite variables, therefore model checking is a viable technique for formal verification in this domain.

2.2 Modelling Methodology

In this section, we give a brief overview of our previous approach to generate formal models of PLC programs. Our goal is to be able to handle diverse input and output formalisms, i.e. PLC programs written in different PLC languages and the model formats of various verification tools. Supporting several verification tools means that we can exploit their advantages in terms of simulation facilities, property specification methods, and verification techniques. For this reason, we designed a generic *intermediate model* (IM) [1], representing the behaviour of

the verified programs. This model is based on a network of automata[1] representing the control flow graph (CFG) of the PLC program. The IM is a structure $\mathcal{M} = (A, I)$, where A is a set of automata, I is a set of synchronizations. Each automaton is a tuple $A_i = (S, T, V, s_0, V_0)$, where L is a set of states (locations), T is a set of guarded transitions, V is a set of typed variables, l_0 is the initial state and V_0 is the initial value of the variables. A synchronization $I_i = (t, t') \in I$ connects two transitions that should be fired together. We allow maximum one synchronization connected to a transition t.

This IM is similar to the formalism defined in [12], but without clocks for time representation and with slightly different synchronization semantics (see [1] for more details). It provides easy adaptability to a big variety of model checking tools.

Thus, our transformation workflow consists of two main steps: first, the source code of the program is transformed into the IM using the known specialities of the PLC environment (like the scan cycle). The result is essentially a CFG. Next, the IM is translated to the inputs of the verification tools, like NuSMV, UPPAAL or BIP. This approach makes the transformation easy to extend as it decouples the input and output languages. The reader can find further information and more details on this modelling methodology in our report [1].

2.3 Complexity Issues

Our former experiments have shown that the presented modelling approach can be used successfully for transforming ST code to the input languages of various model checker tools. However, verification was seldom successful.

Each base object of the UNICOS library can contain hundreds of variables, not to mention programs consisting of multiple objects. We take as an example an often-used base object of the library that is a behavioural representation of an actuator (e.g. valves, heaters, motors) driven by digital signals. This object is implemented in ST as a function block, calling several other functions. With 60 input variables (of which 13 are parameters and several integers), 62 output variables, and 3 timer instances, this object is representative of other UNICOS objects in terms of size and complexity.

Therefore it is unavoidable to apply advanced algorithms to reduce the task of the model checker. We apply various reduction and abstraction techniques in the different steps of the transformation chain. We designed the intermediate model so that reduction and abstraction techniques are easily applicable on it, and all the model checkers can benefit from the effect of the reductions. The next section is dedicated to these reduction methods. In Section 4, this base object is used to evaluate our reduction techniques.

[1] Here we target PLC code without interrupts, thus the CFG can be represented by one single automaton. If interrupts are considered, building the product automaton would not be effective.

3 Reduction Techniques

The requirements to be verified are often complex, containing multiple temporal logic operators and involving a large number of variables, which limits the set of possible reduction techniques. Therefore we have chosen to use property preserving reduction methods. (We also apply non-preserving abstractions for specific purposes such as modelling timers, however, that is not within the scope of this paper.) These algorithms aim at preserving only those behaviours of the system that are relevant from the specification point of view. By using property preserving techniques, the meaning of the model is not modified with regard our assumptions, thus these reductions do not induce any false results that would make the verification much more difficult.

Our reductions are applied to the intermediate model that corresponds to the control flow graph of the PLC program. (Most of these reduction techniques are general for any CFG, not just for the CFGs of PLC programs.) In a PLC the requirements need to be checked only at the beginning and at the end of the PLC cycles, but not in the intermediate states, as the transient values are not written to the physical outputs of the PLC. As a consequence, the order of the variable assignments is not important, as long as the result at the end of the PLC cycle will be the same. Thus, we know that the requirements are only to be checked at specific locations of the CFG.

We do not consider concurrency problems. A PLC code without interrupts is usually single threaded, therefore concurrency problems cannot arise.
In the rest of this section, we present our reductions techniques:

- a new heuristic *cone of influence reduction* algorithm adapted for models representing CFGs (Sect. 3.1),
- heuristic *rule-based reduction* techniques to support the cone of influence algorithm (Sect. 3.2), and
- a *mode selection* method which allows the developer to fine-tune the verification by setting the operational mode to be verified (Sect. 3.3).

3.1 Cone of Influence

Cone of influence (COI) [13] is one of the most powerful property preserving reduction techniques. Its main idea is to identify which part of the model is relevant for the evaluation of the given requirement. The unnecessary parts of the model can be removed without affecting the result. This can help to handle the state space explosion problem.

NuSMV has a built-in cone of influence implementation that can reduce the verification time drastically. However, in some cases we experienced slow verification even when the COI algorithm could theoretically reduce the model size to trivial. By analysing this implementation we found out that this reduction technique could be much more powerful if it were *applied to our intermediate model directly*, where the structure of the CFG is known, before it is transformed to a general state-transition system.

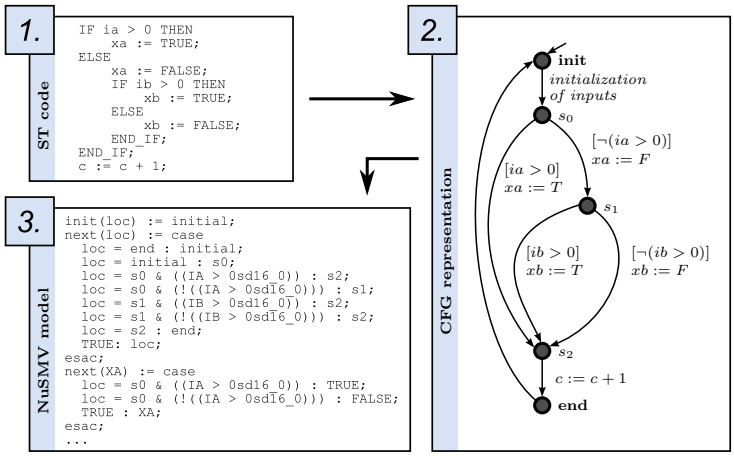


Fig. 1. Example for control flow graph representation of an ST code in NuSMV

In higher level models there is usually more knowledge about the modelled system, therefore the reductions are generally more efficient if they are applied to a higher abstraction level. In this case, the COI algorithm can benefit from the structure information present in the intermediate model, but missing from the generated NuSMV model. Thus, we developed a simple, yet powerful new heuristic cone of influence algorithm, which outperforms the COI method built in NuSMV for our models. To explain the idea behind our approach, first we give an overview of the COI implementation of NuSMV. Afterwards, we introduce our method, and then compare them.

Representing a control flow graph in NuSMV. Before discussing the COI implementation of NuSMV, the method of transforming the CFG represented by the intermediate model to the input language of NuSMV has to be introduced. The input language of NuSMV represents a Mealy machine, whose states comprise the set of possible valuations of every defined state variable [14, 15]. In order to create a NuSMV model corresponding to a control flow graph, a natural way of modelling is to express every variable with a dedicated state variable, and to represent the current location in the CFG with an additional state variable (we call it variable *loc*). This variable *loc* encodes the structure of the CFG along with the guard expressions on its transitions. The example on Fig. 1 shows an extract from a simple ST code, the corresponding CFG and its NuSMV representation.

The COI Implementation of NuSMV. The COI algorithm of NuSMV has no public documentation. However, as this tool is released under the LGPL license, it is possible to analyse its source code.

Algorithm 1: ConeOfInfluence

input : $\mathcal{M}$: network of automata, Q: requirement
output : true, iff COI modified the model

$\mathcal{L} \leftarrow$ `UnconditionalStates`$(\mathcal{M})$;
$D \leftarrow$ `NecessaryVariables`$(\mathcal{M}, \mathcal{L}, Q)$;
Removal of all variables in $V \setminus D$, with their guards and assignments;
return $V \setminus D \neq \emptyset$;

At the creation of the NuSMV's internal COI model, the dependencies of each variable are computed (function `coiInit`). A variable v *depends on* every variable v' used for the computation of v. (E.g., if the next state of v is defined with a case block in the NuSMV model, all the variables and values will be in the set of dependencies.) Then, a transitive closure is computed for all the variables occurring in the formula to be evaluated. If a variable x is necessary for some reason, all the variables presented in the assignment block of x will be necessary too. Therefore, it is trivial that according to the NuSMV implementation, the variable *loc* is necessary for a variable v, if it is assigned at least once in the CFG, because this assignment is guarded by a location of the CFG. As *loc* is necessary, all the variables in its next-state relation definition will be necessary too which means all variables taking place in guards are necessary. Thus none of the variables used in guards can be eliminated by the COI of NuSMV.

Our COI. We observed that usually it would be possible to reduce the model by removing conditional branches that do not affect the variables in the requirement. In the example shown in Fig. 1, none of the conditional branches is necessary if only variable c is used in the requirement, therefore the variables *ia* and *ib* could be eliminated as well.

Of course, guards can affect the control flow, therefore it is not possible to eliminate all the guards. Here we propose a simple heuristic working well for PLC programs, because of their general properties discussed in this section. The base idea is that there are states (locations) in the CFG that are "unconditional", i.e. all possible executions go through them.
This COI variant consists of three steps (formally: see Algorithm 1):

1. identification of unconditional states,
2. identification of variables influencing the evaluation of the given requirement,
3. elimination of non-necessary variables.

In the following part of this section, these three steps are introduced.

Identification of unconditional states. The first step of our COI reduction is to identify the set $\mathcal{L}$ of unconditional states. To ensure the correctness of the algorithm, it is necessary that $\mathcal{L}$ does not include conditional states.

To identify the unconditional states, we defined a measure for the states. A *trace* is a list of states $(s_1, s_2, \ldots, s_n)$, where s_1 is the initial state of the automaton, s_n is the end state of the automaton, and there is a transition between each (s_i, s_{i+1}) state pairs. Let $F(s)$ be the fraction of possible traces[2] going through a state s. It is known that all the traces go trough the initial state s_0, therefore $F(s_0) = 1$. For the rest of the states, F can be calculated based on the incoming transitions.

Let $I_T(s) \in 2^T$ depict the set of incoming transitions to state s, and mark the set of outgoing transitions from state s as $O_T(s) \in 2^T$. Let $I_S(t) \in S$ be the source state of transition t. With this notation, the formal definition of $F(s)$ is the following:

$$F(s) := \begin{cases} 1 & \text{if } s = s_0, \\ \sum_{t \in I_T(s)} \frac{F(I_S(t))}{|O_T(I_S(t))|} & \text{otherwise} \end{cases}. \tag{1}$$

After calculating F for each state, it is easy to compute the set of unconditional states: $\mathcal{L} = \{s \in S : F(s) = 1\}$.

Notice, that the intermediate model can contain loops (beside the main loop representing the PLC cycle) due to loops in the input model. In this case, we handle all states in the loop as conditional states (without computing F for them). It has to be noted too that the UNICOS base objects do not contain any loops, and it is also common for other industrial PLC programs.

Identification of necessary variables. The goal of the second step is to collect automatically all the variables necessary to evaluate the given requirement (see Algorithm 2). Let D be the set of necessary variables. It is trivial that every variable in the requirement should be in D. After that, for each variable assignments that modify a variable in D, the variables in the assignment will also be added to the set D. Furthermore, the guard dependencies should also be added to D, i.e. all the variables that can affect the execution of the analysed variable assignment. If the set of D grew, all the assignments should be checked again.

In the following, we define a function $\mathcal{A}_T: T \to 2^V$, which gives all the variables that can affect the execution of variable assignments on transition t.

First, a supporting function $\mathcal{A}_S: S \to 2^V$ is defined which gives all the variables necessary to determine if state s will be active or not during an execution. This function can benefit from the previously explored unconditional states $\mathcal{L}$, as it is known that no variable is necessary to decide if they can be activated during an execution. Thus, for every state $s \in S$, the function $\mathcal{A}_S(s)$ is the following:

$$\mathcal{A}_S(s) := \begin{cases} \emptyset & \text{if } s \in \mathcal{L} \\ \bigcup_{t \in I_T(s)} \mathcal{A}_T(t) & \text{if } s \notin \mathcal{L} \end{cases}. \tag{2}$$

It means that for the unconditional states, the set of affecting variables is empty, as a consequence of their definition. If a state is not unconditional, the

[2] Here we do not consider cycles inside the CFG beside of the cycle corresponding to the main cycle.

set of variables affecting that this state is active or not is the set of variables affecting the firing of all its incoming transitions.
For every transition $t \in T$, the function $\mathcal{A}_T(t)$ is the following:

$$\mathcal{A}_T(t) := \mathcal{A}_S\left(I_S\left(t\right)\right) \cup \bigcup_{t' \in O_T(I_S(t))} \{\textit{variables in guard of } t'\}. \tag{3}$$

It means that the firing of transition t is affected by the variables that influences its source state ($I_S(t)$) and by the variables in the guard of t. Furthermore, it is also influenced by the variables used in the guard of transitions that can be in conflict with t, i.e. the guard variables of $t' \in O_T(I_S(t))$.

Elimination of non-necessary variables. In the second step, the set D of necessary variables is determined. In the last step, all the variables not in D should be deleted. Also, to ensure the syntactical correctness, all the guards and variable assignments containing variables $v \notin D$ should be deleted. They do not affect the evaluation of the requirement, otherwise they should be in set D.

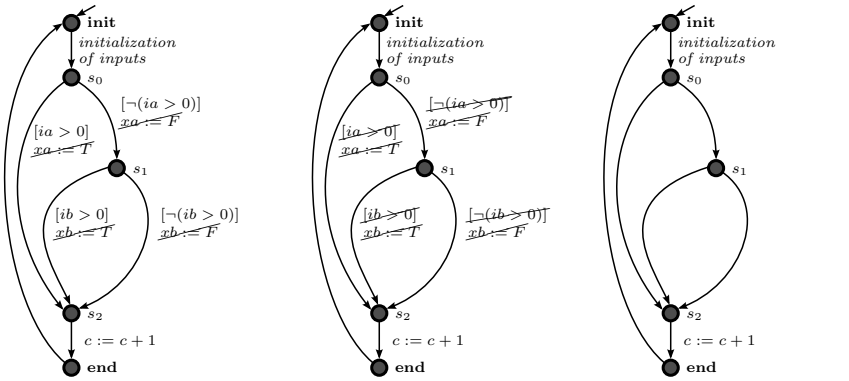
Difference between our COI and the COI of NuSMV. The main difference between our COI and the COI implementation of NuSMV is in the handling of conditional branches. If there is a conditional branch in the CFG, but the variables in the requirement are not affected by this choice, there is no need for the variables in its guard. This difference can be observed in Fig. 2, which shows the CFG introduced in Fig. 1, after applying the different COIs, and assuming that only variable c is used in the requirement (e.g., $\mathsf{EF}\ c < 0$). The COI of NuSMV can identify that the variables xa and xb are not used to compute c, therefore they will be removed. But the variables ia and ib are used in guards, thus they are kept (see Fig. 2(a)).

Our COI algorithm can detect that because xa and xb are deleted, the ia and ib variables can be deleted too, because those guards do not affect c, i.e. no matter which computation path is executed between locations s_0 and s_2, the assignment of variable c will be the same (see Fig. 2(b)).

Algorithm 2: NecessaryVariables

input : $\mathcal{M}$: network of automata, $\mathcal{L}$: set of unconditional states,
Q: requirement
output : D : set of necessary variables

$D \leftarrow \{\textit{variables in } Q\}$;
repeat
  foreach *variable assignment* $\langle v := Expr \rangle$ *on transition* t **do**
    if $v \in D$ **then**
      `// adding assignment and guard dependencies`
      $D \leftarrow D \cup \{\textit{variables in } Expr\} \cup \mathcal{A}_T(t)$;
until D *is not changed*;
return D;



(a) CFG after NuSMV's COI

(b) CFG after our COI

(c) CFG after our COI (without the eliminated parts)

Fig. 2. Example comparing our and the NuSMV's cone of influence algorithm

3.2 General Rule-based Reductions

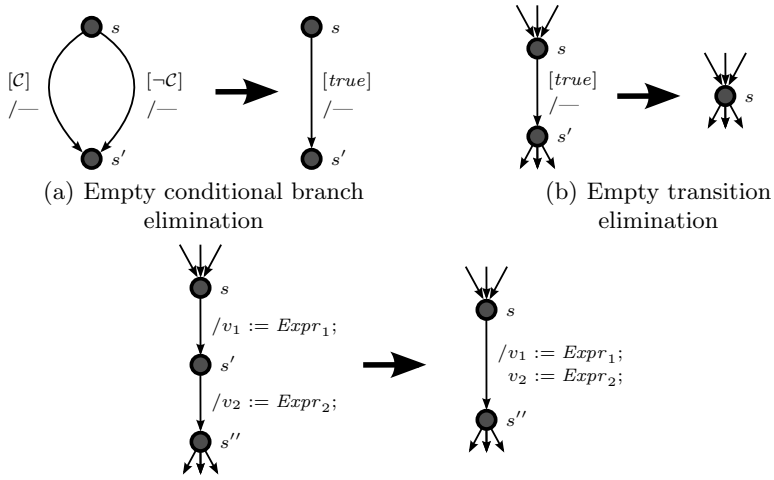
Our cone of influence implementation only eliminates variables and the connected variable assignments and guards, without modifying the structure of the CFG. Therefore, the resulting model of the COI often contains empty transitions, unnecessary states, etc. We identified the frequent situations when reduction can be applied and we developed heuristic, rule-based reductions that make the model smaller and easier to verify.

This kind of CFG reductions are not new, they are used for example in numerous compilers to simplify the machine code and to improve the performance. We refer the reader for details to [16].

Similar reductions can be applied for some particularities in the input source codes. For example, in our UNICOS examples it is common to have an array of Boolean values, where some bits are always false. Using simple heuristics, this situation can be identified and instead of creating an unnecessary variable to store that bit, it can be substituted with the constant false value.

All the general reductions presented are defined by a matching pattern and the modification that should be performed, thus they are defined as a set of graph transformations. Our reductions can be categorized as follows:

– **Model simplifications.** The aim of these reductions is to simplify the intermediate model without reducing its potential state space (thus without eliminating states or variables). This group includes elimination of unnecessary variable assignments or transitions, and simplification of logical expressions. These reductions do not reduce the model themselves, we created them to support the other reduction techniques. For example, if the model contains an empty conditional branch (due to for example another reduc-



(a) Empty conditional branch elimination

(b) Empty transition elimination

(c) Variable assignment merging rule

Fig. 3. Example reduction rules

tion), it can be removed without changing the meaning of the model. This rule is illustrated with the Fig. 3(a).

- **Model reductions.** These methods can reduce the potential state space by eliminating states and variables. This group includes heuristics to merge transitions or states, to eliminate variables that have always the same constant value, etc. For example, if a transition has no guard, no synchronization, and no variable assignment (which can be a side effect of the COI reduction), then the two states on the end of the transition can be merged and the transition can be deleted. This rule is illustrated with Fig. 3(b).
- **Domain-specific reductions.** We can benefit from the assumption we made at the beginning of Section 3 that variables are only checked at the end of the PLC cycle. Our two main domain-specific reductions are the following:
  - *Transition merging.* Two consecutive transitions can be merged if they represent a sequence in the CFG and their variable assignments are not in conflict, i.e. their order does not affect each other's value. Formally, $v_a := E_a$ and $v_b := E_b$ are not in conflict if $v_a \notin \mathit{Vars}(E_b)$, $v_b \notin \mathit{Vars}(E_a)$, and $v_a \neq v_b$, where $\mathit{Vars}(E)$ means all variables used in expression E. This can easily be extended to two set of variable assignments, thus it can be reapplied to merged transitions. Fig. 3(c) illustrates this reduction rule.
  - *Variable merging.* Two variables can be merged if they always have the same value at the end of the PLC cycles. While creating two variables for the same purpose can be considered as a bad programming practice, it is a common pattern in the source code of our systems. This feature is used to improve code readability by assigning a meaningful name to a bit of an input array (for example `PFailSafePosition` instead of `ParReg01[8]`).

Traceability. It has to be noted that not all the previously mentioned reduction techniques are property preserving in themselves. For instance, if two variables v and w are merged, thus w will be deleted, the properties containing w are not possible to evaluate. For this reason, we generate a mapping throughout the reductions that contains all the variable substitutions. Based on this mapping, the necessary aliases can be generated for the NuSMV model. For example, an alias w can be added to the model having always the same value as v, as it is known that they always contain the same value at the end of the PLC cycles. Aliases do not modify the size of the state space or the complexity of the model checking.

Workflow. As discussed in the beginning of Section 3.2, the COI algorithm can enable the rule-based reductions. Therefore, after the COI algorithm, all the reduction techniques introduced in this section will be applied, which can enable other reductions or other possible variable removal for the COI algorithm. Therefore we implemented the reductions in an iterative manner. First, the COI is executed. Then, all the possible reductions are applied. If the COI or one of the rule-based reductions was able to reduce the model, all the reductions are executed again. This iterative workflow is described formally in Algorithm 3. (The function $\texttt{Reduce}(r, \mathcal{M})$ applies reduction r on the model $\mathcal{M}$ and returns true, iff the reduction modified the model.)

Example. A simple example is shown here illustrating our reduction workflow. If our COI is applied to the Fig. 1 CFG example, then Fig. 2(c) CFG will be obtained. If the reductions presented on Fig. 3(a) and Fig. 3(b) are applied on it, the result will be the simple Fig. 4 CFG.

Algorithm 3: Reductions

input : $\mathcal{M}$: model, Q: requirement

bool *changed*;
repeat
  changed $\leftarrow$ $\texttt{ConeOfInfluence}(\mathcal{M}, Q)$;
  foreach $r \in \{$*rule-based reductions*$\}$ **do**
    changed $\leftarrow$ $\texttt{Reduce}(r, \mathcal{M}) \vee$ *changed*;
until *changed* = *false*;

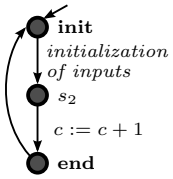


Fig. 4. Example CFG after our COI and the reductions

3.3 Mode Selection

Our initial motivation is to verify UNICOS base objects, the basic building modules of the control systems in CERN. Each base object is generic and can be adapted for the specific uses by setting some parameters. Parameters are input variables that are constant during the execution, as they are hard-coded into the PLC application.

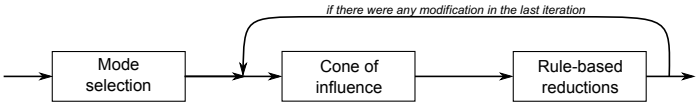


Fig. 5. Overview of the reduction workflow

A significant part of our real requirements have assumptions on the parameters, like "if parameter p_1 is true, then always true that ...". For these requirements, we can use general models and express the parameter configuration in the temporal logic formula, or by adding invariants to the model. However, better performance can be achieved if we encode the configuration in the model itself by replacing the parameter with its constant value. This way, the developers can select the operational mode of the object on which the verification should be performed, i.e. they can fine-tune the verification to their needs. This method is applied only once, before all the other reductions.
The advantages of this method are the following:

- There is no need to create a variable for the fixed parameter as it is substituted with a constant.
- The rule-based reductions can simplify the expressions. For example, if there is a guard $[p_1 \wedge p_2 \wedge p_3]$ in the CFG and p_1 is fixed to false, the guard will be always false and can be replaced by a constant false.
- The simplification of the expressions can help the cone of influence reduction to remove the unnecessary variables. E.g., if p_2 and p_3 are only used in the guard $[p_1 \wedge p_2 \wedge p_3]$, and p_1 is fixed to false, the variables p_2 and p_3 can be removed. Using the COI in NuSMV or if the fixed parameter is expressed through invariants, this reduction cannot be done.

Now the complete reduction workflow can be summarized. After the transformation of the source code into the intermediate model, the given parameters are fixed to the required values. Then the cone of influence reduction and the rule-based reductions are performed iteratively (as seen in Fig. 5). It is important to note, that since every reduction rule deletes something from the model upon firing, they cannot "reduce" the model infinite times, thus the iteration cycle will stop in finite time. The transformation workflow extended with the reduction techniques can be seen on Fig. 6.

4 Evaluation

In this section, we show measurement results to evaluate our solution. For the evaluation we used the base object introduced in Section 2.3. The effect of our methods on the state space and the run time reduction are introduced. Furthermore, these measurements illustrate, how the different reduction techniques can complement each other.

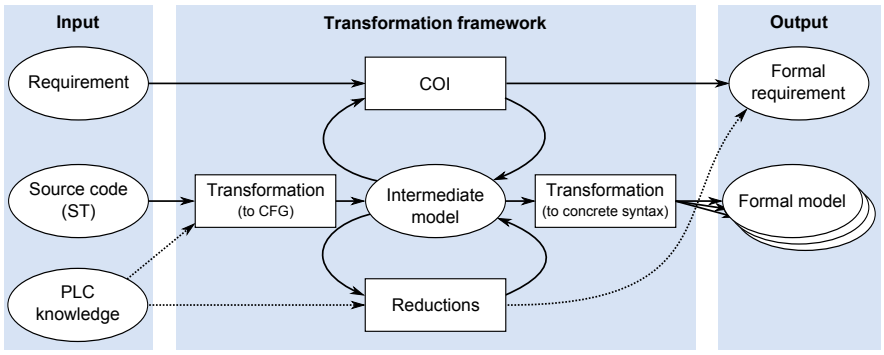


Fig. 6. Overview of the modelling approach

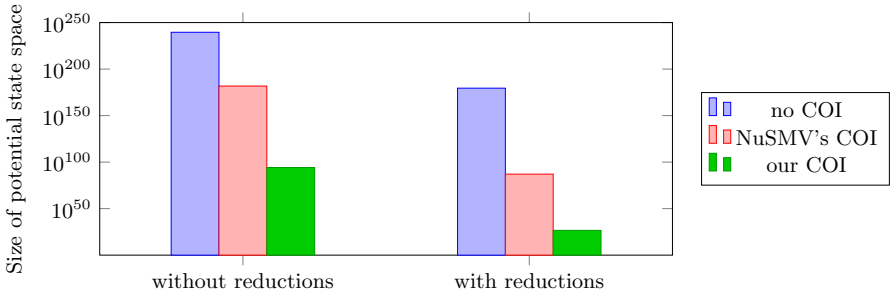


Fig. 7. Measurements comparing the different COI solutions

State space. The model generated from the base object is big, as its potential state space (PSS, the number of possible values of all the variables) contains $3.8 \cdot 10^{239}$ states without any reductions. Fig. 7 shows measurements about the size of the potential state space without and with our rule-based reductions, and with different COI reductions using a real-life requirement without mode selection. As can be seen, NuSMV's cone of influence can reduce the state space significantly, however our implementation provides much better results. For example, if the rule-based reductions are enabled, the size of the PSS is $1.2 \cdot 10^{87}$ with NuSMV's COI, and $4.3 \cdot 10^{26}$ with our own COI implementation.

Verification run time. The difference between the COI implementations can also be observed on the run time of the verification. Table 1 shows verification run time measurements and measurements about the internal BDD (binary decision diagram) data structures of NuSMV that represent the state space. In these cases, three different real-life requirements were evaluated. Req. 1 is an LTL expression with form of $\mathsf{G}(\alpha \,\mathsf{U}\, \beta)$, Req. 2 and 3 are CTL safety expressions ($\mathsf{AG}(\alpha)$), Req. 4 is a complex LTL expression ($\mathsf{G}((\alpha \wedge \mathsf{X}(\beta \,\mathsf{U}\, \gamma)) \rightarrow \mathsf{X}(\beta \,\mathsf{U}\, \delta))$) describing a real requirement coming from the developers. In the requirements the Greek

letters represent Boolean logical expressions containing multiple (usually 1–5) variables. Without our reductions (even if NuSMV's COI is used), none of them could be executed in a day, thus these reductions are inevitable. These measurements show that by using our COI implementation, the verification run time can be reduced by 1–3 orders of magnitude compared to the COI of NuSMV. The same reduction can be observed in the number of allocated BDD nodes (#Node). The reduction in the peak number of live BDD nodes (#PNode) is smaller, but significant. These measurements show the efficiency of our method.

Table 1. Requirement evaluation measurements

	no reduct.+	our reductions + NuSMV COI			our reductions + our COI		
Req.	NuSMV COI	Runtime	#PNode	#Node	Runtime	#PNode	#Node
1	—	896 s	$8.8 \cdot 10^5$	$1.8 \cdot 10^8$	2.5 s	$2.2 \cdot 10^5$	$1.1 \cdot 10^6$
2	—	1,250 s	$9.9 \cdot 10^5$	$8.8 \cdot 10^8$	19.0 s	$4.6 \cdot 10^5$	$1.4 \cdot 10^7$
3	—	19,300 s	$3.4 \cdot 10^6$	$1.6 \cdot 10^{10}$	1,440 s	$1.3 \cdot 10^6$	$1.6 \cdot 10^9$
4	—	649 s	$9.0 \cdot 10^5$	$5.5 \cdot 10^8$	2.3 s	$2.2 \cdot 10^5$	$9.2 \cdot 10^5$

5 Conclusion and Future Work

This paper presents a solution to make the formal verification of real PLC programs possible by extending automatic model generation with property preserving reduction techniques. These reduction techniques are part of a general methodology based on an intermediate model suitable for transforming the formal models of PLC programs into the input format of different verification tools.

Our results show that by using cone of influence reduction (tailored to our specific application domain) and simple rule-based reduction techniques allows us to apply model checking even to complex real PLC programs, such as the base objects of the CERN's control systems. We have also shown that the effectiveness of the cone of influence algorithm can be significantly improved by performing the reductions directly on the intermediate model, and by exploiting relevant domain-specific knowledge. Moreover, further fine-tuning can be obtained with our proposed method of handling parameter configurations for operational mode selection, by adapting the model to the scenario to be checked.

Our future plans comprise the integration of the tool and methodology in the UNICOS development process. In addition, extending and optimizing the abstraction techniques are ongoing work.

References

1. Darvas, D., Fernández, B., Blanco, E.: Transforming PLC programs into formal models for verification purposes. Internal note, CERN (2013) http://cds.cern.ch/record/1629275/files/CERN-ACC-NOTE-2013-0040.pdf.

2. IEC 61131: Programming languages for programmable logic controllers. (2013)
3. Rausch, M., Krogh, B.: Formal verification of PLC programs. In: Proc. of the American Control Conference 1998. (1998) 234–238
4. Bauer, N., Engell, S., Huuck, R., Lohmann, S., Lukoschus, B., Remelhe, M., Stursberg, O.: Verification of PLC programs given as sequential function charts. In: Integration of Software Specification Techniques for Applications in Engineering. Volume 3147 of LNCS. Springer (2004) 517–540
5. Canet, G., Couffin, S., Lesage, J.J., Petit, A., Schnoebelen, P.: Towards the automatic verification of PLC programs written in Instruction List. In: Proc. of Int. Conf. on Systems, Man, and Cybernetics 2000, Argos Press 2449–2454
6. Pavlović, O., Ehrich, H.D.: Model checking PLC software written in function block diagram. In: International Conference on Software Testing. (2010) 439–448
7. Soliman, D., Frey, G.: Verification and validation of safety applications based on PLCopen safety function blocks. Control Engineering Practice **19**(9) (2011) 929–946
8. Gourcuff, V., de Smet, O., Faure, J.M.: Improving large-sized PLC programs verification using abstractions. In: 17th IFAC World Congress. (2008)
9. Lange, T., Neuhäußer, M., Noll, T.: Speeding up the safety verification of programmable logic controller code. In: Hardware and Software: Verification and Testing. Volume 8244 of LNCS. Springer (2013) 44–60
10. Biallas, S., Brauer, J., Kowalewski, S.: Counterexample-guided abstraction refinement for PLCs. In: Proc. of 5th International Workshop on Systems Software Verification, USENIX Association (2010) 2–12
11. Blanco, E., et al.: UNICOS evolution: CPC version 6. In: 12th ICALEPCS. (2011)
12. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: SFM-RT 2004., Revised Lectures. Volume 3185 of LNCS., Springer Verlag (2004) 200–237
13. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
14. Cavada, R., Cimatti, A., Jochim, C.A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., Tchaltsev, A.: NuSMV 2.5 User Manual. FBK-irst. (2011)
15. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Computer Aided Verification. Volume 2404 of LNCS. Springer (2002) 359–364
16. Cooper, K.D., Torczon, L.: Engineering a Compiler. Second edn. Morgan Kaufmann Publishers Inc. (2012)