Simplification of numeric variables for PLC model checking

Conference Paper · November 2021

DOI: 10.1145/3487212.3487334

5 authors, including:

Ignacio D. Lopez-Miguel
CERN
2 PUBLICATIONS 1 CITATION
SEE PROFILE

Borja Fernández Adiego
CERN
19 PUBLICATIONS 173 CITATIONS
SEE PROFILE

Jean-Charles Tournier
CERN
31 PUBLICATIONS 390 CITATIONS
SEE PROFILE

Enrique Blanco Viñuela
CERN
65 PUBLICATIONS 435 CITATIONS
SEE PROFILE

Simplification of numeric variables for PLC model checking

Ignacio D. Lopez-Miguel*
Borja Fernández Adiego
Jean-Charles Tournier
Enrique Blanco Viñuela
European Organization for Nuclear Research (CERN),
Beams Department
Geneva, Switzerland

Juan A. Rodriguez-Aguilar
Artificial Intelligence Research Institute (IIIA-CSIC)
Bellaterra, Spain

ABSTRACT

Software model checking has recently started to be applied in the verification of programmable logic controller (PLC) programs. It works efficiently when the number of input variables is limited, their interaction is small and, thus, the number of states the program can reach is not large. As observed in the large code base of the CERN industrial PLC applications, this is usually not the case: it thus leads to the well-known state-space explosion problem, making it impossible to perform model checking. One of the main reasons that causes state-space explosion is the inclusion of numeric variables due to the wide range of values they can take. In this paper, we propose an approach to discretize PLC input numeric variables (modelled as non-deterministic). This discretization is complemented with a set of transformations on the control-flow automaton that models the PLC program so that no extra behaviours are added. This approach is then quantitatively evaluated with a set of empirical tests using the PLC model checking framework PLCverif and three different state-of-the-art model checkers (CBMC, nuXmv, and Theta), showing beneficial results for BDD-based model checkers.

CCS CONCEPTS

• **Software and its engineering → Model checking**.

KEYWORDS

programmable logic controller (PLC), model checking, verification, predicate abstraction, control-flow automaton (CFA)

ACM Reference Format:
Ignacio D. Lopez-Miguel, Borja Fernández Adiego, Jean-Charles Tournier, Enrique Blanco Viñuela, and Juan A. Rodriguez-Aguilar. 2021. Simplification of numeric variables for PLC model checking. In *19th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '21), November 20–22, 2021, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3487212.3487334

*Corresponding author, ignacio.david.lopez.miguel@cern.ch

1 INTRODUCTION

Software model checking is a very popular approach to prove the correctness of programs and to find bugs. It has recently started to be applied in the verification of programmable logic controller (PLC) programs and it has been proven to be beneficial for different applications, where discrepancies between the specifications and the programs were found that were not detected by applying testing techniques [3, 14].

Application of model checking to PLC programs is limited to the number of input variables, their interaction and to the number of states the program can reach. However, PLC programs at CERN[1] usually include several variables of different nature, such as Booleans, integers, reals, words, and arrays. In this situation, the well-known state-space explosion may occur, which leads to the impossibility to perform model checking [9].

The aim of this paper is to simplify numeric variables in order to reduce the state space in the context of PLC programs. This could be done by using predicate abstraction [19], i.e., constructing a new abstract representation of the model from a set of predicates over the different variables [17]. However, regardless of the abstraction used, more behaviours are included into the program, requiring to perform refinement when a found counterexample is not correct [28].

Counterexample-guided-abstraction refinement (CEGAR) is a popular approach to introduce abstraction in the verification and to refine it according to the obtained counterexample. However, this approach is not done in a unique way, but there exist different CEGAR techniques and various implementations depending on the model checker [33].

In contrast, taking advantage of the open-source PLC model checking framework PLCverif[2] developed at CERN, this paper introduces a family of transformations to simplify input numeric variables (modelled as non-deterministic at the beginning of each PLC cycle in PLCverif) that do not add extra behaviours and are agnostic of the model checker used. The transformations consist in a discretization of the domain of the original variables, which are associated with specific structural patterns of the formalized model (control-flow automaton in PLCverif). In order not to lose information during these transformations, the control-flow automaton is extended according to the structural patterns.

In addition to reducing the state space of the original model, the proposed approach provides richer counterexamples: since the

[1]Organisation europóene pour la recherche nucléaire (European Council for Nuclear Research).
[2]It is publicly available under https://gitlab.com/plcverif-oss

transformed variables represent intervals, the counterexamples produced are defined as a set of inequalities, describing a whole space of counterexamples instead of specific values for each variable.

Several experiments implementing the proposed transformations have been conducted in order to quantify the impact of applying the proposed approach in the performance of verification of PLC programs. A performance boost of up to 300x was achieved for the used state-of-the-art binary-decision-diagram-based (BDD-based) model checker. In addition to this, in some cases this model checker timed out with the original model, but it was able to finish in less than a second with the transformed one.

The remainder of this paper is organized as follows: Section 2 gives an overview of the previous works related to model checking applied to PLC program verification, as well as other techniques to simplify numeric variables, in particular, predicate abstraction and interval analysis. Section 3 provides a brief introduction to PLCverif, the system formalization it implements (control-flow automaton), and the different model checkers it currently supports. Section 4 presents the main contribution of the paper by defining a family of transformations that can be applied to simplify numeric variables according to a set of structural patterns. Section 5 uses the transformations from Section 4 in order to introduce a general method to simplify any numeric variable with some given limitations. A series of experiments are presented in section 6, where the impact of using the suggested transformations is quantified in the three model checkers used by PLCverif (CBMC, nuXmv, and Theta). Finally, section 7 concludes the paper explaining the results and proposing possible future lines of research.

2 RELATED WORK

During the last decades, model checking has been not only studied in academia but also successfully applied in several industrial domains [20]. While some work has been done on PLC based code verification [16], model checking has been mostly applied in other industries [2, 4–6, 13, 18, 21, 24, 26].

Some tools relying on formal methods to verify PLC programs have been developed, such as Arcade.PLC [3], MODCHK [25], PLCInspector [32], and PLCverif [10]. Arcade.PLC is a framework for the verification of PLC programs developed by Aachen University and consists of a simulator for different PLC languages, a model-checker and a static analysis interface. MODCHK is a graphical tool developed by VTT Technical Research Centre of Finland that verifies function block based application logics, generating the necessary input files to run an external model checker, visualizing the counterexample graphically. PLCInspector is a tool that directly mines linear temporal logic specifications and data invariants from PLC programs. PLCverif is an industrial framework developed at CERN relying on external model checkers to verify PLC programs automatically.

PLCverif was the tool selected to run the experiments of this paper since PLC code is translated automatically to the input language of three external model checkers. This allows us to apply the transformations presented in this paper directly in the PLC code manually and get the transformed model for the different model checkers automatically. In addition to this, we have already expertise with PLCverif since it was used to validate various real-life PLC projects at CERN [14] and it was internally developed.

Model simplification to improve the performance of PLC model checking has already been the target of other works, such as [23], where a set of transformations is proposed so that bounded model checking (BMC) techniques based on satisfiability modulo theories (SMT) improve their performance. Also, reduction techniques to simplify a control-flow-automaton have been studied and implemented in PLCverif, such as cone of influence, which removes all the variables that are not relevant for the verified property [11].

Predicate abstraction [19] is a common technique used to reduce the state space in model checking. It creates a new abstract representation of the model from a set of predicates over the different variables [17]. However, once a counterexample is found in the abstract domain, it has to be validated since more behaviours are included into the model [28]. If the counterexample is not valid, a refinement of the abstraction is performed, leading to the so-called counterexample-guided abstraction refinement (CEGAR) [33]. However, this refinement process may not terminate unless some conditions are given [27], such as overapproximating numeric variables in order to disprove non-terminating branches [22]. Our approach presented in this paper tries to avoid these limitations by proposing a method that does not require refinement.

The work done in [1] introduces a novel technique to contract the domain of a set of n variables in an n dimensional space in order to prune the state space when performing bounded model checking. The main issue is that the contracted domain can contain cases where the property is violated and cases where it is satisfied. In comparison, in the approach presented in this paper, the created intervals can only contain either satisfied or violated cases due to the way they are constructed.

Finally, in [31], boundaries of numeric variables are set manually so that the model checker does not need to explore all values. This approach could be as well combined with our suggested technique since it could help to reduce the number of discrete values that are needed.

3 BACKGROUND

This section briefly presents the PLCverif framework along with its formalism used to represent PLC program as it is at the basis of the transformations presented in this paper. This section also introduces the three model checkers currently supported by PLCverif.

PLCverif is a customisable and extensible, plugin-based framework developed at CERN to support formal verification of PLC programs. It directly translates PLC code and a given set of requirements automatically to the input format of various model checkers, which are then executed. Their output is presented to the user in a convenient, easy-to-understand format. This workflow is shown in Figure 1. More details can be found in [12, 30].

The mathematical model used by PLCverif to formalize a given program is a network of control-flow automata (CFA) representing the control flow graph of the PLC program. Input variables are modelled as non-deterministic since the external system to which the PLC is connected is not modelled, and, hence, these inputs could have any possible value.

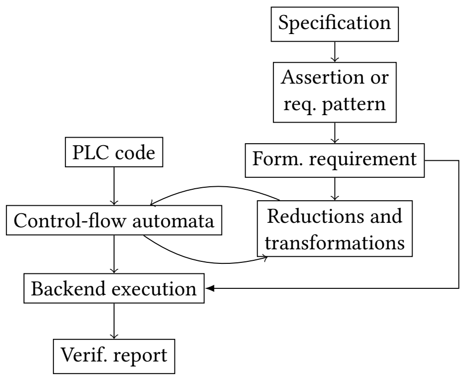


Figure 1: Verification process workflow using PLCverif. Given a specification and PLC code, after various intermediate steps, it generates a report specifying if the specification is satisfied or violated. More information can be found in [30].

PLCverif uses different state-of-the-art symbolic model checkers based on different model checking algorithms:

- nuXmv [7] is a BDD-based model checker. BDDs efficiently construct and represent sets of states compactly. It has the limitation that numeric variables need to be encoded as a set of Booleans.
- CBMC [8] is a SAT-based bounded model checker. It is based on unwinding loops a given number of times or using k-induction in order to obtain a SAT formula representing all the behaviours of the program. Due to this limitation unwinding loops, it does not explore the whole state space.
- Theta [29] is a CEGAR-, SMT-based model checker. Satisfiability Modulo Theories (SMT) solvers check the satisfiability of first-order logic formulas by using a SAT solver and a theory solver (T-solver). The later includes a set of logics that check the feasibility of the given solution by the SAT solver.

4 A FAMILY OF TRANSFORMATIONS

The purpose of this section is to introduce a series of transformations based on structural patterns present in CFAs that widely appear in the large code base of PLC programs at CERN. The goal of the transformations presented in this section is that numeric variables that are modelled as non-deterministic at the beginning of the PLC cycle can be discretized without losing or adding information. The approach focuses on CFAs without loops or with bounded loops. Based on the CERN industrial PLC applications, this is not considered to be a limitation since unbounded loops, or with a conditional exit, rarely appear.

The different transformations are introduced in an increasing order of complexity, starting with specific cases and ending with a generalization of them. Thus, we design transformations for the following structural patterns:

- Conditional pattern. It finds simple conditional statements in the transition guards where an input variable x is compared with a constant value a_i. The transformation discretizes the domain of x according to the values a_i.
- Assignment pattern. It finds transitions where assignments take the form of $x \leftarrow cx + b$, where x is an input variable, and c and b are constant values. It propagates these assignments across the CFA making sure no information is lost by duplicating the necessary locations (CFA nodes).
- Generalization of the conditional pattern. Making use of interval arithmetic, an input variable which appears in any type of conditional statement is transformed.
- Generalization of the assignment pattern. Similarly, interval arithmetic allows handling any assignment in order to transform an input variable following the steps from the assignment pattern transformation.

In section 5, these transformations will then be combined in a specific order to transform any input numeric variable.

For each of the presented transformations, the pattern is structurally characterized and the corresponding mathematical transformation is explained.

It is assumed that the CFA to be simplified has been previously reduced using cone of influence [11], i.e., only the relevant variables for the assertion are kept.

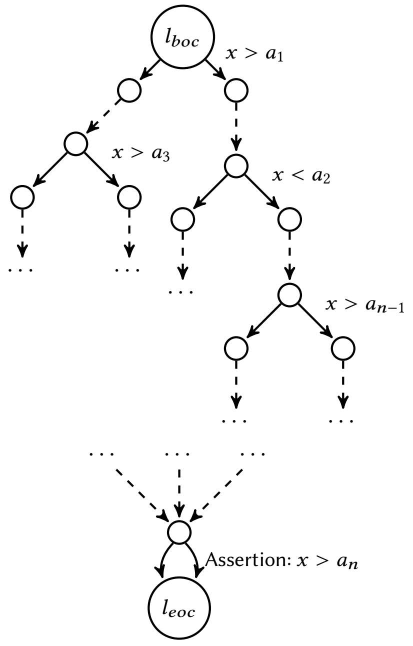


Figure 2: CFA conditional pattern.

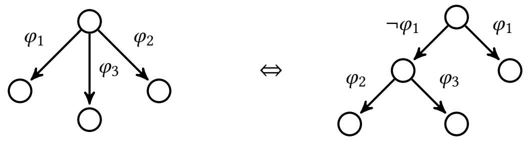


Figure 3: Transformation of a node with three exiting edges to nodes with only two exiting edges.

4.1 Conditional pattern

4.1.1 Characterization. A CFA contains a conditional pattern if there exists an input numeric variable (non-deterministic at the beginning of the PLC cycle) x such that x is part of the assertion (or part of the variable dependency graph of any of the variables forming the assertion), there is no assignment $x_i \leftarrow f(x, ..., x_n)$, where x_i is any variable of the CFA, and all the expressions where x appears are comparisons of this variable with a constant a_i.

The CFA for a variable matching this pattern is represented in Figure 2. Variable x is the guard for different transitions and there are no assignments where this variable is used. The CFA has been depicted with only two edges exiting each location, but it could be more than two. However, they can always be recursively reduced to two, as shown in Figure 3, where φ_i represents a given expression.

4.1.2 Transformation. Let us define the set of expressions where variable x appears $\Phi = \{\varphi_1, \varphi_2, ..., \varphi_n\}$. These expressions are a comparison between x and a constant a_i. Thus, they can be split into its operator $op(\varphi_i) \in \{<, >, \leq, \geq, =, \neq\}$ and its constant $cte(\varphi_i) \in \mathbb{R}$. So we can define the set of all unique constants:

$$A = \{a_1, a_2, ..., a_{n_A}\} \subseteq D = \{x \in \mathbb{R} : x_{min} \leq x \leq x_{max}\} \subseteq \mathbb{R},$$

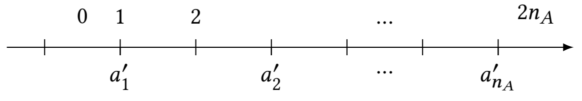
where D is the domain of variable x, x_{min} and x_{max} are the minimum and maximum values for the variable x according to its type, and $n_A \leq n$ is the number of unique constants.

We define the ordered set $A' = \{a'_1, ..., a'_{n_A}\}$, such that $a'_1 \leq a'_2 \leq ... \leq a'_{n_A}$.

We can now introduce the variable transformation:

$$f : D \longrightarrow \{0, 1, ..., 2n_A\}$$

$$x \longmapsto f(x) = \begin{cases} 0, & \text{if } x < a'_1 \\ 1, & \text{if } x = a'_1 \\ 2, & \text{if } a'_1 < x < a'_2 \\ 3, & \text{if } x = a'_2 \\ ... & \\ 2n_A, & \text{otherwise.} \end{cases}$$



Let I be the set of intervals with lower and/or upper limit values from A'. We can transform this interval into a set of finite values according to the previous variable transformation:

$$g : I \longrightarrow N \subset \{0, 1, ..., 2n_A\}$$

$$I_i \longmapsto g(I_i) = \{f(inf(I_i)), ..., f(sup(I_i))\} \cap$$
$$\cap \{f(min(I_i), ..., f(max(I_i))\}$$

Since the intervals that are transformed are the same than the ones used to define the variable transformation, it would not be necessary to transform the variable into $2n_A$ values, but in a smaller or equal number.

For each unique constant, a_i, a boundary in the variable transformation function, f, is created. This boundary can take three different shapes, depending on how the domain of the original variable, x, is split, i.e.:

1 unique operator

Operator	Boundary
$<$	$)[$
$>$	$]($
$=$	$) \cdot ($
$\neq$	$) \cdot ($
$\leq$	$]($
$\geq$	$)[$

2 unique operators

Operators	Boundary
$<, >$	$) \cdot ($
$<, \geq$	$)[$
$<, \leq$	$) \cdot ($
$>, \geq$	$) \cdot ($
$>, \leq$	$]($

Table 1: Types of boundaries resulting from the different operators.

$$(1)\ )[\quad \Leftrightarrow \quad \begin{cases} x < a'_i \\ x \geq a'_i \end{cases} \qquad (3)\ ) \cdot (\quad \Leftrightarrow \quad \begin{cases} x < a'_i \\ x = a'_i \\ x > a'_i \end{cases}$$

$$(2)\ ](\quad \Leftrightarrow \quad \begin{cases} x \leq a'_i \\ x > a'_i \end{cases}$$

Depending on the number and type of unique operators contained in the expressions associated to each constant, the corresponding transformation will be one of the previously-shown three boundaries as displayed in Table 1. The combination of three or more different operators results in $) \cdot ($ since this combination will always contain two operators with $) \cdot ($.

Let us define the following function, which assigns to each constant a type of boundary for a given Φ:

$$h_\Phi : a \in A' \longrightarrow \{1, 2, 3\},$$

$$a \longmapsto h_\Phi(a) = \begin{cases} 1, & \text{if } \forall \varphi \in \Phi | cte(\varphi) = a : op(\varphi) \in \{<, \geq\} \\ 2, & \text{if } \forall \varphi \in \Phi | cte(\varphi) = a : op(\varphi) \in \{>, \leq\} \\ 3, & \text{otherwise.} \end{cases}$$

So the general variable transformation for a given Φ is

$$f : D \longrightarrow \{0, 1, ..., n_A\}$$

$$x \longmapsto I_x = f(x) = Card(\{a \in A' : a \leq x \wedge h_\Phi(a) = 1\}) +$$
$$Card(\{a \in A' : a < x \wedge h_\Phi(a) = 2\}) +$$
$$2 \cdot Card(\{a \in A' : a < x \wedge h_\Phi(a) = 3\}) +$$
$$Card(\{a \in A' : a = x \wedge h_\Phi(a) = 3\})$$

$Card(A)$ is the cardinality of the set A. Notice that the interval transformation remains as it was previously defined.

It is important to highlight that, due to the fact that the discretized representation of the original variables groups sets of values into intervals, the counterexamples resulting from the transformed CFA represent a whole set of counterexamples. They are defined as a set of inequalities derived from the definition of the intervals. This also applies to the rest of patterns since it will be shown that they are based on this one. In contrast, model checkers usually only give one counterexample.

4.2 Assignment pattern

4.2.1 Characterization. A CFA contains an assignment pattern if there exists a numeric variable x that is part of the variable dependency graph of any of the variables forming the assertion, and the assignments to this variable take the shape $x \leftarrow cx + b$, where b and c are constant values. This variable must as well be non-deterministic at the beginning of the PLC cycle. Furthermore, the

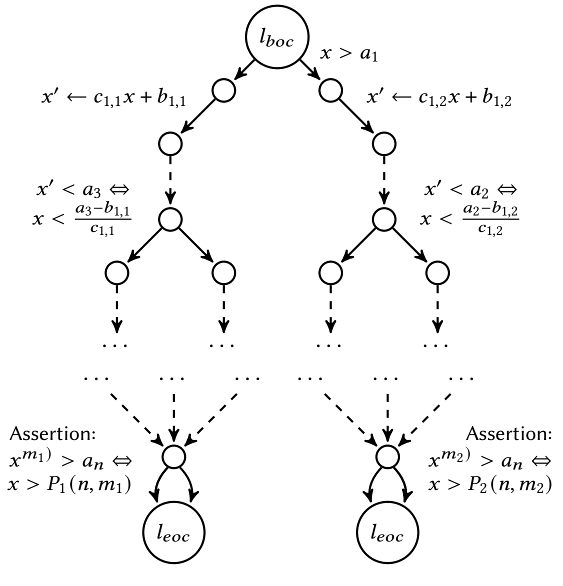

l_{boc}
$x > a_1$
$x' \leftarrow c_{1,1}x + b_{1,1}$
$x' \leftarrow c_{1,2}x + b_{1,2}$
$x' < a_3 \Leftrightarrow$
$x < \frac{a_3-b_{1,1}}{c_{1,1}}$
$x' < a_2 \Leftrightarrow$
$x < \frac{a_2-b_{1,2}}{c_{1,2}}$
Assertion:
$x^{m_1)} > a_n \Leftrightarrow$
$x > P_1(n, m_1)$
Assertion:
$x^{m_2)} > a_n \Leftrightarrow$
$x > P_2(n, m_2)$
l_{eoc}
l_{eoc}

Figure 4: CFA transformation for the assignment pattern. The end location l_{eoc} and other nodes have been duplicated.

transition guards remain the same than with the conditional pattern, i.e. just a comparison between the variable and a constant. Also, the CFA can only contain loops with a fixed number of iterations. The assignment $x \leftarrow y$ is as well considered in this pattern, where y is any other variable.

Now we distinguish two cases:

(1) Unique node sequences. For a given variable x, there is only one path in the CFA with assignments. The assignments can be removed recursively, modifying the conditional statements accordingly. This leads to a transformed CFA with a conditional pattern.
(2) Non-unique node sequences. For a given variable x, there are multiple paths with assignments. A node duplication is needed in order to keep all the behaviours of the model. After duplicating the nodes, the assignments can be removed recursively and the assertions modified accordingly so that for each sequence of unique assignments the conditional pattern transformation can be applied.

The second case is a generalization of the first one, so only the second one is presented.

4.2.2 Transformation. The main goal of this transformation is to simplify the CFA so that the conditional pattern transformation can be applied.

To start with, assignments that take the form $x \leftarrow y$, where y is any other variable, are removed and x is simply replaced by y in all expressions that follow this assignment in the CFA.

The end-of-cycle location, l_{eoc}, will be duplicated as many times as the number of different paths (starting at the beginning-of-cycle location, l_{boc}, and ending at l_{eoc}) where there are assignments to x. This way it will be possible to modify the assertion so that it is possible to build the whole trace of a counterexample. Each of these paths will have a unique sequence of assignments, hence assignments can be removed recursively. Figure 4 shows a CFA where l_{eoc} has been duplicated and the assertion has been modified.

Not only the l_{eoc} will be duplicated, but all its parent locations in order to achieve unique sequences of assignments.

In order to define this transformation correctly, we need to introduce the *Assignments* function, which assigns to each transition $t \in T$ the set of expressions E that represent assignments done to a certain variable x_i in that transition:

$$Assignments : T \rightarrow E \subset \{\{x_i \leftarrow cx_i + b\} : c, b \in \mathbb{R}\} \cup$$

$$\cup \{\{x_i \leftarrow x_j\} : j \in (1, N_{var})\},$$

where N_{var} is the number of variables.

For each transition, t_i, there is a start location, $Start(t_i)$, and an end location, $End(t_i)$.

Formally, for a given location, l_i, the number of duplications, $n_{i,dup}$, that are needed is the number of all unique product of sequences of transitions, (t_1, t_2, ..., t_n), with its assignments sequences ($Assignments(t_1)$, $Assignments(t_2)$, ..., $Assignments(t_n)$), such that $Start(t_1) = l_{boc}$ and $End(t_n) = l_i$. If there are two non-equal sequences of transitions but their associated sequence of assignments is the same, there is no duplication needed. Note that $(Assignments(t_i)) = (Assignments(t_i), \emptyset)$ and that a sequence of assignments for a given path can also be empty if there are no assignments.

All the transitions sequences between l_{boc} and a given location, l_i, in the original CFA must be the same than the union of all the transitions sequences between l_{boc} and the duplicated nodes of l_i in the transformed CFA. Furthermore, the sequence of assignments belonging to any possible sequence of transitions starting at l_{boc} and ending at a given duplicated node is only one. With these conditions, we make sure that it is possible to build a whole trace of a counterexample and that we do not lose information.

Once all the duplications are done, for each assignment its value is carried over to the next locations. Then, the CFA will no longer have assignments and all the comparisons will be between x and a constant. Therefore, this is the situation of the conditional pattern, so its above-introduced transformation can be applied.

For simplicity, for the CFA of Figure 4, it is assumed that all values multiplying x are positive ($c_{i,j} > 0$). Therefore, values P_1 and P_2 correspond to the following expressions, where the recursive replacement appears and m_i is the number of assignments to the variable x in that path and n the number of comparisons with x in the whole CFA:

$$P_1(n, m_1) = \frac{a_n - \sum_{i=1}^{m_1-1} \left(b_{i,1} \prod_{j=i+1}^{m_1} c_{j,1} \right) - b_{m_1,1}}{\prod_{i=1}^{m_1} c_{i,1}}$$

$$P_2(n, m_2) = \frac{a_n - \sum_{i=1}^{m_2-1} \left(b_{i,2} \prod_{j=i+1}^{m_2} c_{j,2} \right) - b_{m_2,2}}{\prod_{i=1}^{m_2} c_{i,2}}$$

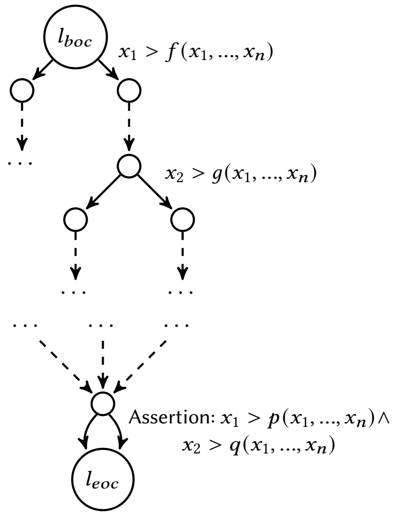


Figure 5: CFA for the generalization of the conditional pattern.

In case there exists a negative $c_{i,j}$, we would need to take into account the sign of $\prod_{i=1}^{k} c_{i,j}$ to modify the expressions of the assertion accordingly (a similar transformation is obviously needed for $\leq$ and $\geq$):

$$x^{k)} > a_j \begin{cases} x > P(j,k), & \text{if } \prod_{i=1}^{k} c_i > 0 \\ x < P(j,k), & \text{if } \prod_{i=1}^{k} c_i < 0 \end{cases}$$

$$x^{k)} < a_j \begin{cases} x < P(j,k), & \text{if } \prod_{i=1}^{k} c_i > 0 \\ x > P(j,k), & \text{if } \prod_{i=1}^{k} c_i < 0 \end{cases}$$

In case there is a $c_{i,j} = 0$, there will be an assignment $x^{i)} \leftarrow b_{i,j}$. Thus, it will be possible to change all the assignments and guards of the child locations after the assignment $x^{i)} \leftarrow b_{i,j}$ to its corresponding deterministic value $b_{i,j}$.

4.3 Generalization of the conditional pattern

4.3.1 Characterization. This pattern is an extension of the conditional one, where the expressions of the guards are a comparison between the input variable and a function of other variables. A generic CFA can be seen in Figure 5.

4.3.2 Transformation. Let us take the variables on which x_i depends and the variables that depend on x_i:

$$\{x_1, ..., x_n\}$$

The simplification described in the conditional pattern is applied to all these variables at once taking into account only the expressions with constant functions. This means that all the operations and constants are collected and a unique variable transformation function, f, for all variables is created. This transformation is applied to all the variables of the list, resulting in a list of transformed variables:

$$\{I_{x_1}, ..., I_{x_n}\}$$

Each value of the transformed variable I_{x_i} represents an interval of the original variable x_i. Since we need to transform expressions where two or more variables are involved, we will need interval arithmetic.

Let us define a family of binary operations, $\star$, where the operands are closed intervals:

$$[a_1, a_2] \star [b_1, b_2] = \{a \star b : a \in [a_1, a_2], b \in [b_1, b_2]\}$$

This operation could be any, such as addition, product, etc.

If the intervals are not closed, the operation can also be defined by replacing the upper and/or lower bounds by its supremum and/or infimum values, and let that part of the interval open. For example,

$$[a_1, a_2) \star (b_1, b_2] = \{a \star b : a \in [a_1, a_2), b \in (b_1, b_2]\}$$

If $\star$ is monotone in each operand on the intervals, the extreme values occur at the endpoints of the operand intervals. Thus,

$$[a_1, a_2] \star [b_1, b_2] = [\min\{a_1 \star b_1, a_1 \star b_2, a_2 \star b_1, a_2 \star b_2\}, \\ \max\{a_1 \star b_1, a_1 \star b_2, a_2 \star b_1, a_2 \star b_2\}]$$

In case that any of the intervals is not closed, *min* (*max*) should be replaced by *inf* (*sup*) and let the interval open for the lower and/or upper bound.

Examples of operations with closed intervals:

- Addition:
$$[a_1, a_2] + [b_1, b_2] = [a_1 + b_1, a_2 + b_2]$$
- Subtraction:
$$[a_1, a_2] - [b_1, b_2] = [a_1 - b_2, a_2 - b_1]$$
- Multiplication:
$$[a_1, a_2] \cdot [b_1, b_2] = [\min\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}, \\ \max\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}]$$

Examples of operations with open intervals:

- Addition:
$$(a_1, a_2] + [b_1, b_2] = (a_1 + b_1, a_2 + b_2]$$
$$(a_1, a_2] + [b_1, b_2) = (a_1 + b_1, a_2 + b_2)$$
- Subtraction:
$$(a_1, a_2] - [b_1, b_2] = (a_1 - b_2, a_2 - b_1]$$
$$(a_1, a_2] - [b_1, b_2) = (a_1 - b_2, a_2 - b_1]$$
- Multiplication (for simplification we denote $min = \min\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}$ and $max = \max\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}$):
$$(a_1, a_2] \cdot [b_1, b_2] = \begin{cases} (min, max], \text{if } min \in \{a_1b_1, a_1b_2\} \\ [min, max), \text{if } min \notin \{a_1b_1, a_1b_2\} \end{cases}$$
$$(a_1, a_2] \cdot [b_1, b_2) = \begin{cases} [min, max), & \text{if } min = a_2b_1 \\ (min, max], & \text{if } max = a_2b_1 \\ (min, max), & \text{otherwise.} \end{cases}$$
$$(a_1, a_2) \cdot [b_1, b_2] = (\min\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}, \\ \max\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\})$$

Therefore, a conditional statement, $x_1 < g(x_1, x_2, ..., x_n)$, can be transformed into a disjunction of conditions provided the transformation function $f : x \longmapsto I_x, x \rightarrow f(x)$. Indeed x_1 is transformed to $I_{x_1} = f(x_1)$ and function g can be transformed to its equivalent with interval arithmetic, $\bar{g} : (I_{x_1}, ..., I_{x_n}) \longmapsto G \subset \{0, 1, ..., n_A\}$. The codomain of $\bar{g}$ is formed by sets and not by singles values because the result of an interval operation can result in an interval containing several intervals. Besides, the same interval operation with different interval operands can result in the same interval.

Therefore, if the result of applying $\bar{g}$ to two given values has n possible values, an auxiliary non-deterministic variable, w with n values will be added to decide which value takes place. If this variable is encoded with m Boolean variables, it must hold that $2^m \geq n > 2^{m-1}$. Thus, provided the transformation function f, then this auxiliary variable, w is defined as follows:

$$w = \begin{cases} 0, & \text{if } f(g(x_1, ..., x_n)) = 0 \\ \dots & \\ n, & \text{otherwise.} \end{cases}$$

In case that $I_{x_1} = \bar{g}(I_{x_1}, I_{x_2}, ..., I_{x_n})$, another non-deterministic auxiliary variable, w_1, will be added to decide whether $x_1 < g(x_1, x_2, ..., x_n)$, which is defined as follows:

$$w_1 = \begin{cases} True, & \text{if } x_1 < g(x_1, x_2, ..., x_n) \\ False, & \text{otherwise.} \end{cases}$$

Notice that w_1 is not defined if $I_{x_1} \neq \bar{g}(I_{x_1}, I_{x_2}, ..., I_{x_n})$.

Now the original conditional statement can be written as:

$$\Big[\bigvee_{\substack{(A_1,...,A_n): \\ \forall I \in \bar{g}(A_1,...,A_n): I_{x_1} < I}} (I_{x_1}, ..., I_{x_n}) = (A_1, ..., A_n) \Big] \vee$$

$$\Big[\bigvee_{\substack{(A_1,...,A_n): \\ I_{x_1} \in \bar{g}(A_1,...,A_n)}} (I_{x_1}, ..., I_{x_n}) = (A_1, ..., A_n) \wedge w < I_{x_1} \Big] \vee$$

$$\Big[\bigvee_{\substack{(A_1,...,A_n): \\ I_{x_1} \in \bar{g}(A_1,...,A_n)}} (I_{x_1}, ..., I_{x_n}) = (A_1, ..., A_n) \wedge w = I_{x_1} \wedge w_1 \Big]$$

Notice that I_{x_1} is a variable and can have different values. Hence, the previous expression must be instantiated with each of its values, and all of them joined together with a disjunction. Now all variables and expressions are transformed.

In order to define the counterexamples after applying this transformation, the model checker needs to specify the value of the auxiliary variables as it would be done with any other non-deterministic variable. These values are only relevant if the auxiliary variables are defined for the rest of values of the other variables. Then, this results in a set of counterexamples as for the previous patterns.

4.4 Generalization of the assignment pattern

4.4.1 Characterization. This pattern is an extension of the assignment one, where the assignments can be functions of other variables. A generic CFA of a program that contains this pattern would look as the one shown in Figure 6. This pattern is limited to CFAs with bounded loops.

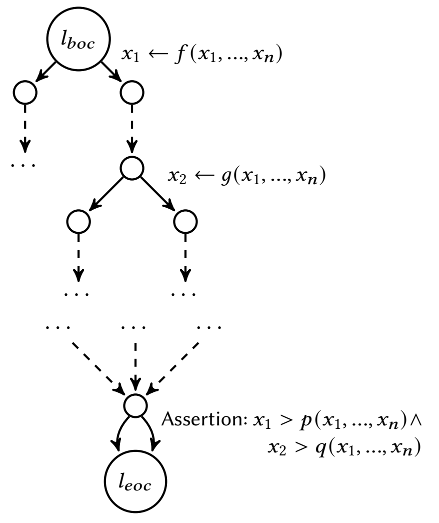

l_{boc}
$x_1 \leftarrow f(x_1, ..., x_n)$
$x_2 \leftarrow g(x_1, ..., x_n)$
Assertion: $x_1 > p(x_1, ..., x_n) \wedge$
$x_2 > q(x_1, ..., x_n)$
l_{eoc}

Figure 6: CFA for the generalization of the assignment pattern.

4.4.2 Transformation. The idea is completely analogous to the one used for the generalization of the conditional pattern, but including as well the location duplication step from the assignment pattern in order to have unique sequences of assignments. This means that all the possibilities are calculated and the necessary auxiliary variables to decide which values take place are added.

When an assignment takes the form $x_1 \leftarrow x_2$, there are two options:

(1) creating a path for each value of x_2; or
(2) replacing x_1 by x_2 in the child nodes.

The second option is preferred due to its simplicity with respect to the first one. As well, in the experimental results, the second option will be taken since the resulting CFA and the process to obtain it are much simpler.

Let us consider the assignment $x_i \leftarrow g(x_1, x_2, ..., x_n)$. For each possible value of $\bar{g}$, $I_{\bar{g}}$, there will be a transition with a condition and an assignment. The condition is

$$\Big[\bigvee_{\substack{(A_1,...,A_n): \\ \bar{g}(A_1,...,A_n)=\{I_{\bar{g}}\}}} (I_{x_1}, ..., I_{x_n}) = (A_1, ..., A_n) \Big] \vee$$

$$\Big[\bigvee_{\substack{(A_1,...,A_n): \\ \{I_{\bar{g}}\} \in \bar{g}(A_1,...,A_n)}} (I_{x_1}, ..., I_{x_n}) = (A_1, ..., A_n) \wedge w = g_i \Big]$$

The assignment is then trivial for each $I_{\bar{g}}$ ($I_{x_i} \leftarrow I_{\bar{g}}$).

Now it is possible to apply the simple assignment pattern transformation to finally remove all assignments.

Similar to the generalization of the conditional pattern, the counterexample returned by the model checker contains the values for the auxiliary variables, which are only relevant when they are defined according to the other variables.

5 GENERAL TRANSFORMATION

Finally, we introduce a general method that takes advantage of the family of transformations presented previously to transform any numeric variable. These transformations have to be carried in a specific order so that the result is correct.

5.1 Characterization

The general transformation allows to transform any numeric variable modelled as non-deterministic at the beginning of the PLC cycle that is relevant for the assertion. During the previous section when all the transformations have been introduced, the target variables of the assignment pattern transformations have been limited to non-deterministic variables at the beginning of the cycle. This is due to the fact that deterministic variables at the beginning of the cycle carry their value over cycles, which makes it impossible to remove assignments.

Nevertheless, if deterministic variables are converted to non-deterministic at the beginning of the cycle, the original behavior of the code is kept. However, more behaviors are added. This occurs since the non-deterministic new variables can take any possible value, whereas the original ones can only take a subset of them given by a specific program.

Therefore, if the assertion is satisfied with the non-deterministic variables, it would mean that the original assertion is as well satisfied. In contrast, if a counterexample is found, extra steps to check that the counterexample is valid would be needed, such as the ones presented in [15] or using CEGAR.

5.2 Transformation

The steps that have to be applied in order to be able to transform any numeric variable with the restrictions previously explained are as follows:

(1) Apply the transformation from the generalization of the assignment pattern:
   (a) *Duplicate the locations for all kinds of assignments.*
   (b) *Simplify simple assignments ($x \leftarrow ax + b$ or $x \leftarrow y$).*
(2) Apply the transformation from the conditional pattern.
(3) Apply the transformation from the generalization of the assignment pattern:
   (a) *Duplicate the locations of assignments with non-constant functions.*
   (b) *Simplify assignments with non-constant functions.*
(4) Apply the transformation from the generalization of the conditional pattern.
(5) Apply the transformation from the assignment pattern.

The first step is to duplicate all the needed locations so that every path contains a unique sequence of assignments. The conditional pattern cannot be applied before this transformation since the variables might change across the CFA.

Once the locations are duplicated, one can simplify the assignments from the assignment pattern ($x \leftarrow ax + b$ or $x \leftarrow y$). As in the previous step, the conditional pattern cannot be applied before this step since variables will change.

However, once this is done, it is possible to apply the transformation from the conditional pattern since there are no more simple assignments.

At this point, we focus on complex assignments and conditions. Similarly to the previous steps, we start by simplifying assignments. In this case, the transformation from the generalization of the assignment pattern has to be used since only complex assignments are left. Therefore, we need to duplicate the necessary locations in order to include the corresponding conditions for each assignment. This will simplify the complex assignments into constant assignments.

In order to simplify the complex conditions, we take advantage of the transformation from the generalization of the conditional pattern. At this point, there are no more original variables, but transformed variables.

For this reason, we just need to apply the transformation from the assignment pattern to simplify the constant assignments that were left. Now, there are no more assignments and the conditions are just composed of transformed variables. Therefore, the transformation is finalized.

6 EXPERIMENTAL RESULTS

A list of four different CFAs are selected to quantify the benefit of applying the presented transformations. For a given CFA, the performance can vary significantly depending whether the assertion is satisfied or violated. Therefore, for each of the tests, a case where the assertion is violated and another one where the assertion is satisfied are included.

The selected CFAs were created in order to target one or various patterns. The summary of them is as follows:

- Test 1: Simple conditional pattern, cf. Figure 7.
- Test 2: Simple conditional pattern and simple assignment pattern, cf. Figure 8.
- Test 3: Simple conditional pattern and general assignment pattern, cf. Figure 9.
- Test 4: General conditional pattern and general assignment pattern, cf. Figure 10.

For all tests, the corresponding CFA was instantiated 13 times with different variables. Therefore, the assertion the model checkers verify is a conjunction of 13 expressions. This is done in order to have bigger time differences and be able to ignore other terms, such as memory allocation time.

The three model checkers included in PLCverif were taken (nuXmv, Theta, and Theta) to test the proposed approach.

The CFAs (both original and transformed) were given as PLC code to PLCverif, which then translated them into the input language of the different model checkers[3].

The main purpose of testing the transformations on different CFAs and with different model checkers is to get some insight about when it is useful to transform a given CFA for each type of model checker.

The measures used for this objective are the following:

- Total time: it includes the execution time and the time the model checker needs to build the model.
- Speed-up factor: it is the ratio between the time spent on verifying the original model over the simplified one:

$$time_{original} / time_{simplified}$$

The higher this number is, the better the simplification is.

[3]The code used for this section can be found in the public repository *msc-work*.

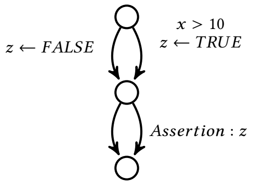


Figure 7: Test 1. CFA original for the simple conditional pattern (violated property).

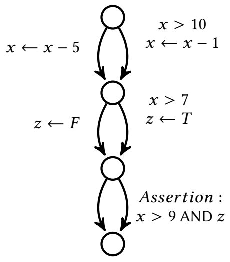


Figure 8: Test 2. Original CFA for the simple conditional pattern and simple assignment (violated property).

		Original Mean (s)	Simplified Mean (s)	Speed-up
nuXmv	Test 1 (viol.)	8.71	0.03	279.23
	Test 1 (sat.)	8.84	0.07	126.27
	Test 2 (viol.)	-	0.6	-
	Test 2 (sat.)	14.13	0.15	96.13
	Test 3 (viol.)	-	653.39	-
	Test 3 (sat.)	-	0.72	-
	Test 4 (viol.)[4]	-	218.13	-
	Test 4 (sat.)[4]	-	166.06	-
Theta	Test 1 (viol.)	0.63	0.62	1.02
	Test 1 (sat.)	0.64	0.7	0.92
	Test 2 (viol.)	0.74	0.97	0.76
	Test 2 (sat.)	1.3	1.09	1.19
	Test 3 (viol.)	1.37	1.25	1.1
	Test 3 (sat.)	68.98	4.13	16.72
	Test 4 (viol.)	3.55	8.04	0.44
	Test 4 (sat.)	18.78	26.72	0.7
CBMC	Test 1 (viol.)	0.22	0.19	1.18
	Test 1 (sat.)	0.22	0.20	1.13
	Test 2 (viol.)	1.83	1.17	1.57
	Test 2 (sat.)	1.96	1.24	1.57
	Test 3 (viol.)	13.35	11.37	1.17
	Test 3 (sat.)	15.15	10.65	1.42
	Test 4 (viol.)	15.43	1317.89	0.01
	Test 4 (sat.)	16.47	579.26	0.03

Table 2: Total time results for the three model checkers on the eight different tests. The time is given as a mean of the 20 times the same code was executed. A speed-up greater than 1 means that the proposed approach achieves better performance than the original model.

In order to have significant results and to avoid any possible outliers, every example is executed 20 times, resulting in all the difference being significant as indicated by the T-tests (not reported in this paper).

All experiments were carried on an AMD Ryzen 7 2700X at 4 GHz with 48GB RAM memory, running Windows 10 Home. The total times for all the tests for all model checkers can be seen in Table 2. There is a clear positive impact of applying the transformations for nuXmv. This was expected as the state space is hugely reduced[5]. This reduction also makes nuXmv verify models that originally timed out.

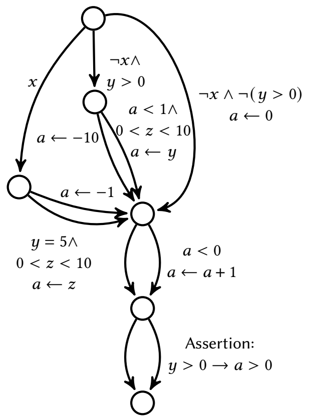


Figure 9: Test 3. CFA original for the simple conditional pattern and general assignment pattern (violated property).

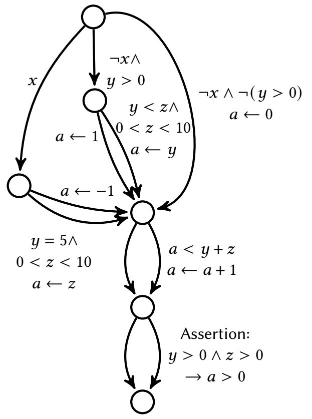


Figure 10: Test 4. CFA original for the general conditional pattern and general assignment pattern (violated property).

With respect to Theta, results are mixed. There is a sign that the simplified version works better than the original one when there is no violation of the property. This is reasonable since, for the original version, Theta may have to discover all relevant predicates, needing to include all possible combinations of the numeric variables if it is not able to find the right abstraction. This will depend on how the heuristic to find predicate abstractions is developed. On the other hand, in the simplified version, this combination is limited by the combination of Booleans. It also seems that if a lot of interval arithmetic is needed to do the suggested transformation (tests 3 and 4), the performance drops. This can be explained by the increased size of the control structure due to the higher number of operations needed to encode the transformation. However, this deterioration

[4]The CFA is only repeated twice, not 13.

[5]As an example, for the test 2 (sat.), the state space of the original model is in ca. 10^{140}, whereas for the transformed version, it is in the order of 10^{23}

is not as big as with CBMC, which might indicate that it could work for other kind of CFAs with similar logic.

Finally, this approach shows a benefit for CBMC when the models are not very complex (test 1, 2 and 3). However, when many extra Boolean variables and operations are needed to encode the transformation, CBMC does not take advantage of this approach[6]. This means that the encoding needs to be further investigated in order to find an efficient way to do it so that the final SAT formula does not explode, but reduces.

Therefore, from the original purpose of this section, the results can be summarized as follows:

- Based on the four different CFAs tested, nuXmv shows a much better performance on the transformed model than on the original model. This indicates that transforming the CFA for nuXmv may always be beneficial.
- Since the most complex CFA showed worse results on the transformed model than on the original one for Theta, there is an indication that the transformations are only useful for Theta if the CFA is simple (no generalization of the conditional and assignment pattern simultaneously). For simpler CFAs, results are mixed. However, the transformations can be useful when the assertion is satisfied.
- Similarly, for CBMC the transformations do not seem to be beneficial when both generalizations are present.

7 CONCLUSION AND FUTURE WORK

In this work, we introduced a family of transformations that can be applied to the formalized model (CFA) of PLC programs according to different structural patterns in order to simplify numeric variables. By taking advantage of this set of transformations, we proposed a general transformation that targets to simplify any numeric variable.

The findings obtained during this work can be summarized as follows based on the previous experimental results:

- It is possible to simplify input numeric variables that are non-deterministic at the beginning of the PLC cycle without adding behaviours in a program with fixed loops.
- Since the discrete values of the transformed variables represent intervals, counterexamples resulting from executing a model checker on the transformed CFA represent a whole set of counterexamples. In contrast, the state-of-the-art model checkers used in this paper only return a single counterexample and not a set of them if no transformation is done.
- Due to the extreme state-space reduction that the presented transformations perform, it is beneficial to use it in BDD-based model checkers, such as nuXmv.
- CEGAR-based model checkers, such as Theta, can take advantage of these transformations when the property to be checked is satisfied. However, when it is violated, the abstraction CEGAR performs can find a valid counterexample faster.
- SAT-based model checkers that include predicate abstraction, such as CBMC, can profit from the transformations in both cases, when the assertion is satisfied and when it is violated. However, due to the extra operations needed to encode the transformation, there is a performance drop when the model includes complex operations (generalization of the conditional and assignment patterns).

This paper presented an efficient way to handle the state-space explosion problem when applying model checking on PLC programs with numerical variables, which opens a number of promising, future research paths:

- The transformation has been done manually changing the PLC code and letting PLCverif produce the corresponding transformed model for the different model checkers. Since it has been shown benefitial for nuXmv, it would be interesting to investigate how to build an algorithm for that and to automate it within PLCverif.
- Since the transformation has been done manually, the computation time to find the patterns and transform the model has not been taken into account. This time is not expected to be significant in comparison to the reduction times for nuXmv, but it should be analysed once it is automatically implemented.
- The translation from the CFA to the model checker input language has not been studied during this work. However, the efficiency of the model checker can vary depending on how this is encoded as it has been seen for CBMC.
- Although the patterns and transformations are general and could be applied to other programming languages, this work has focused on PLC programs. Thus, an analysis of the impact of these transformations in other programming languages could be done, extending the patterns to include other behaviours.
- In order to see in advance if the transformation is going to be beneficial, it would make sense to study, provided a CFA, how the CFA is going to grow, how many transformed variables are going to be created, and how many auxiliary variables are going to be added. This could indicate the feasibility of the transformation.
- The work is limited to programs with fixed loops, thus, it could be studied how to combine the suggested approach with k-induction or other techniques to manage programs with unbounded loops.

ACKNOWLEDGMENTS

IIIA-CSIC contribution was supported by project CI-SUSTAIN funded by Spanish Ministry of Science and Innovation (PID2019-104156GB-I00).

REFERENCES

[1] Mohannad Aldughaim, Kaled Alshmrany, Mohamed Mustafa, Lucas Cordeiro, and Alexandru Stancu. 2020. Bounded Model Checking of Software Using Interval Methods via Contractors.

[2] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. 1999. Météor: A Successful Application of B in a Large Project. In *FM'99 — Formal Methods*, Jeannette M. Wing, Jim Woodcock, and Jim Davies (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–387.

[6]This is also reflected in the number of clauses and variables that the final SAT formula has. For example, for test 4 (violated property), where the original model is faster than the transformed one, the number of variables and clauses for the original model are 83,995 and 628,944 respectively. On the other hand, for the transformed model, the number of variables is 77,235 and the number of clauses is 6,265,224 (10 times more)

[3] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. 2012. Arcade.PLC: A Verification Platform for Programmable Logic Controllers. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, 338–341. http://publications.embedded.rwth-aachen.de/file/3w

[4] A. Bouzafour, M. Renaudin, H. Garavel, R. Mateescu, and W. Serwe. 2018. Model-Checking Synthesizable SystemVerilog Descriptions of Asynchronous Circuits. In *2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. 34–42. https://doi.org/10.1109/ASYNC.2018.00021

[5] Marco Bozzano, Harold Bruintjes, Alessandro Cimatti, Joost-Pieter Katoen, Thomas Noll, and Stefano Tonetta. 2019. COMPASS 3.0. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomas Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 379–385.

[6] Julien Brunel, David Chemouil, Alcino Cunha, and Nuno Macedo. 2018. The Electrum Analyzer: Model Checking Relational First-Order Temporal Specifications. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. Montpellier, France. https://doi.org/10.1145/3238147.3240475

[7] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *CAV*. 334–342.

[8] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004) (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176.

[9] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. 2012. *Model Checking and the State Explosion Problem*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–30. https://doi.org/10.1007/978-3-642-35746-6_1

[10] Daniel Darvas, Borja Fernández Adiego, and Enrique Blanco Viñuela. 2015. PLCverif: A tool to verify PLC programs based on model checking techniques. https://doi.org/10.18429/JACoW-ICALEPCS2015-WEPGF092

[11] Daniel Darvas, Borja Fernández Adiego, András Vörös, Tamás Bartha, Enrique Blanco Viñuela, and Víctor Suárez. 2014. Formal Verification of Complex Properties on PLC Programs, Vol. 8461. 284–299. https://doi.org/10.1007/978-3-662-43613-4_18

[12] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. 2016. *Formal Verification of Safety PLC Based Control Software*. Lecture Notes in Computer Science, Vol. 9681. Springer, 508–522. https://doi.org/10.1007/978-3-319-33693-0_32

[13] Yann Duplouy. 2018. *Applying Formal Methods to Autonomous Vehicle Control*. Theses. Université Paris Saclay (COmUE). https://tel.archives-ouvertes.fr/tel-01960966

[14] Borja Fernández Adiego, Dániel Darvas, Jean-Charles Tournier, Enrique Blanco Viñuela, and Víctor M. González Suárez. 2014. Bringing Automated Model Checking to PLC Program Development – A CERN Case Study. In *Proceedings of the 12th International Workshop on Discrete Event Systems*, Jean-Jacques Lesage, Jean-Marc Faure, José E. Ribiero Cury, and Bengt Lennartson (Eds.). International Federation of Automatic Control, Paris, France, 394–399. https://doi.org/10.3182/20140514-3-FR-4046.00051

[15] Borja Fernández Adiego. [n.d.]. Bringing automated formal verification to PLC program development.

[16] Borja Fernández Adiego, Bhimavarapu Avinashkrishna, Enrique Blanco Viñuela, Daniel Darvas, Yogesh Gaikwad, Gisik Lee, Riccardo Pedica, Ignacio Prieto Diaz, Gyula Sallai, and Sailaraj Sreekuttan. 2018. Applying model checking to critical PLC applications: An ITER case study. (2018), THPHA161. 5 p. https://doi.org/10.18429/JACoW-ICALEPCS2017-THPHA161

[17] Cormac Flanagan and Shaz Qadeer. 2002. Predicate Abstraction for Software Verification. *SIGPLAN Not.* 37, 1 (Jan. 2002), 191–202. https://doi.org/10.1145/565816.503291

[18] Arthur Flatau, Matt Kaufmann, David Reed, David Russinoff, Eric Smith, and Rob Sumners. 2002. Formal Verification of Microprocessors at AMD. (01 2002).

[19] Susanne Graf and Hassen Saidi. 1997. Construction of state graphs with PVS, Vol. 1254. 72–83. https://doi.org/10.1007/3-540-63166-6_10

[20] Marieke Huisman, Dilian Gurov, and Alexander Malkis. 2020. Formal Methods: From Academia to Industrial Practice. A Travel Guide. arXiv:2002.07279 [cs.SE]

[21] Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. 2008. Formal Verification of a Flash Memory Device Driver – An Experience Report. In *Model Checking Software*, Klaus Havelund, Rupak Majumdar, and Jens Palsberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 144–159.

[22] Takuya Kuwahara, Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. 2015. Predicate Abstraction and CEGAR for Disproving Termination of Higher-Order Functional Programs. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 287–303.

[23] Tim Lange, Martin Neuhäußer, and Thomas Noll. 2013. Speeding Up the Safety Verification of Programmable Logic Controller Code. 44–60. https://doi.org/10.1007/978-3-319-03077-7_4

[24] Michael Lowry. 2008. Intelligent Software Engineering Tools for NASA's Crew Exploration Vehicle, Vol. 4994. 28–37. https://doi.org/10.1007/978-3-540-68123-6_3

[25] Antti Pakonen, I Buzhinsky, and K Björkman. 2021. Model checking reveals design issues leading to spurious actuation of nuclear instrumentation and control systems. *Reliability Engineering and System Safety* 205 (2021), 107237. https://doi.org/10.1016/j.ress.2020.107237

[26] Antti Pakonen, Topi Tahvonen, Markus Hartikainen, and Mikko Pihlanko. 2017. Practical Aplications of model checking in the finnish nuclear industry. In *10th International Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technologies*. https://cris.vtt.fi/en/publications/practical-applications-of-model-checking-in-the-finnish-nuclear-i

[27] Sanjit A. Seshia and Susmit Jha. 2015. Synthesis, Verification, and Inductive Learning. (07 2015). https://www.lri.fr/~filliatr/1.9/sri-july-2015/slides/Seshia-VerifSW-WGMeeting.pdf

[28] Cong Tian, Zhenhua Duan, and Nan Zhang. 2012. An efficient approach for abstraction-refinement in model checking. *Theoretical Computer Science* 461 (2012), 76–85. https://doi.org/10.1016/j.tcs.2011.12.014 17th International Computing and Combinatorics Conference (COCOON 2011).

[29] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. 2017. Theta: a Framework for Abstraction Refinement-Based Model Checking. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, Daryl Stewart and Georg Weissenbacher (Eds.). 176–179. https://doi.org/10.23919/FMCAD.2017.8102257

[30] E. Blanco Viñuela, D. Darvas, and V. Molnár. 2020. PLCverif Re-engineered: An Open Platform for the Formal Analysis of PLC Programs. In *Proc. ICALEPCS'19 (International Conference on Accelerator and Large Experimental Physics Control Systems, 17)*. JACoW Publishing, Geneva, Switzerland, 21–27. https://doi.org/10.18429/JACoW-ICALEPCS2019-MOBPP01 https://doi.org/10.18429/JACoW-ICALEPCS2019-MOBPP01.

[31] J. Xiong, X. Bu, Y. Huang, J. Shi, and W. He. 2020. Safety Verification of IEC 61131-3 Structured Text Programs. *IEEE Transactions on Industrial Informatics* 17 (2020), 2632–2640. https://doi.org/10.1109/TII.2020.2999716

[32] Jiawen Xiong, Gang Zhu, Yanhong Huang, and Jianqi Shi. 2020. A User-Friendly Verification Approach for IEC 61131-3 PLC Programs. *Electronics* 9, 4 (2020). https://doi.org/10.3390/electronics9040572

[33] Ákos Hajdu and Zoltán Micskei. 2020. Efficient Strategies for CEGAR-Based Model Checking. *Journal of Automated Reasoning* 64 (2020), 1051–1091. https://doi.org/10.1007/s10817-019-09535-x