Model Checking PLC Programs: Enhancing Formalization for Scalability

Jessica Ravakambinintsoa [1] , Emil Dumitrescu [2], Eric Zamaï[3] and Denis Chalon [4]

***Abstract*—Formal verification of PLC programs requires transforming control logic into a mathematical model. Since PLC languages lack a formal semantics, multiple transformation methods exist, based on various code interpretations. This work introduces a novel approach leveraging Single Static Assignments (SSA) to ensure a faithful PLC code transformation while enhancing model checking scalability. By systematically tracking variable assignments, the method accurately captures execution dependencies and preserves control logic. Experimental results demonstrate promising improvements in verification efficiency.**

I. Introduction

Programmable logic controllers (PLC) are key components in automated systems, overseeing complex processes that demand rigorous testing and validation. Ensuring the correctness of PLC programs is crucial, as the duration and complexity of testing phases, such as Factory Acceptance Tests (FAT), can significantly delay system deployment and increase project costs. Minor design errors or overlooked scenarios may require extensive rework, further prolonging commissioning times and impacting operational efficiency.

The IEC 61131-3 standard defines five PLC programming languages: Instruction List (IL), Structured Text (ST), Sequential Function Chart (SFC), Function Block Diagram (FBD), and Ladder Diagram (LD). While this variety provides flexibility, low-level languages such as IL (deprecated) and certain ST constructs are error-prone. The IEC 61508 standard, which emphasizes system verification for safety assurance, highlights that textual languages offer precision and clarity, whereas graphical languages like FBD and LD facilitate compliance by making program structures visually explicit and easier to verify. Conventional testing and simulation methods help detect design errors but do not guarantee full test coverage, as required by Clause 7 of IEC 61508-3. Formal verification has emerged as a key approach to addressing these challenges. Since the first attempts in 1994, where Ladder programs were analyzed using McMillan's SMV formalism, research efforts have explored various techniques such as *model checking* [1], [3], [4], [6], [7], [11], [12], [15], [18], [20]–[22], [25], [26], [28], *theorem proving* [16], static analysis [5], [6], [8], [14], [23], and more recently, *Satisfiability Modulo Theories (SMT)*.

However, a fundamental scientific issue persists: PLC programming languages, as defined by industrial standards, were not originally designed with formal semantics. This lack of a standardized formal foundation complicates the consistency and applicability of verification methodologies. The process of formal verification requires a formalization step, where each PLC program is translated into a mathematical model through a set of transformation rules. These rules, often embedded inside opaque proprietary tools, vary widely across implementations, leading to inconsistencies in analysis. Consequently, a fundamental challenge arises: (1) ensuring that what we code is accurately simulated, (2) ensuring that what we code is precisely what is formally verified, and (3) ensuring that what we code is faithfully executed on physical devices. While industrial tools partially address (1) and (3), challenge (2) remains a critical gap requiring innovative solutions.

This work focuses on addressing this gap by leveraging model checking with a particular emphasis on NuSMV, a tool that combines a powerful modal logic specification language and efficient verification engines (symbolic and SAT-based). Conducted in collaboration with Schneider Electric, this research aims to enhance the formalization of PLC programs to improve the accuracy (challenge 2) and scalability of model checking approaches. A declarative formalization emerges as particularly promising, demonstrating promising performance metrics, supporting a substantial subset of PLC languages, and aligning closely with NuSMV's symbolic framework.

The following sections present the state of the art in PLC program formal verification and introduce a new formal modeling approach for ST programs, which can be applied to other languages such as LD and/or FBD.

II. State-of-the-art

A. Preliminaries

The model checking approach is defined on a state/transition formal model. The Mealy finite state machine (FSM) model [19] is particularly well-suited, as it can explicitly the system outputs alongside inputs and states. Moreover, Mealy FSMs can represent purely combinational behaviors, which are frequent in PLC programs:

$M = \langle X, S, S_0, O, \delta, \lambda \rangle$, where:

- X is a finite set of Boolean input variables;
- S is a finite set of state variables;
- S_0 is an initial valuation of all variables in S;
- O is a finite set of output variables;
- δ is a set of transition functions, one for each state variable in S;
- λ is a set of output functions, one for each output variable in O;

Bounded integer variables are actually handled natively by the NuSMV model checker via their Boolean representation.

[1]jessica.ravakambinintsoa@se.com / jessica.ravakambinintsoa@insa-lyon.fr
[2]emil.dumitrescu@insa-lyon.fr
[3]eric.zamai@insa-lyon.fr
[4]denis.chalon@se.com

Hence, the without loss in generality, this work focuses exclusively on Boolean variables.

1. **initialize** variable S_0;
2. **for each** period **do**;
3. **read** input variable X;
3. **compute** values of $S : S' := \delta(S, X)$;
4. **assign** values of $O : O := \lambda(S, X)$;
5. **assign** $S := S'$;
6. **end**;

Fig. 1. PLC reactive loop

Each PLC program must be formalized as a model M to enable model checking. Given that PLCs operate in a cyclic, reactive manner, formalization through Finite State Machines (FSMs) appears well-suited. Specifically, Mealy FSMs are considered appropriate, as they naturally integrate inputs, outputs, and state transitions, key characteristics of PLC execution. Unlike Moore FSMs, where outputs depend solely on the current state, Mealy FSMs allow outputs to vary dynamically based on both state and inputs, aligning closely with PLC behavior. Consequently, each PLC cycle is modeled as a single transition in the FSM representation. However, a fundamental requirement of this formalization is ensuring bounded execution within a sampling period, as illustrated in Fig1. This is typically feasible in PLC scheduling but may be compromised by algorithmic constructs found in the ST language, such as unbounded loops, jumps, and variable-bound iterations. The IEC 61131-3 standard prohibits recursion, mitigating some risks, but structured handling of control flow remains critical. Despite its advantages, the FSM-based formalization introduces a trade-off: the granularity of states and transitions must be carefully managed to prevent state-space explosion, which could hinder verification scalability. The remainder of this section provides an overview of previous research on PLC program formalization and its integration with model checking approaches.

B. Related works

It can be noticed that PLC programs are formalized according to various interpretations and verification needs. Two main approaches of PLC program formalization can be observed in the literature:

- **intra-cycle** approach: each line of code is modeled as a state, and the resulting FSM representation embodies a control flow graph.
- **inter-cycle** approach: the program's variables define its state. More specifically, all variables featuring a side effect between two successive PLC cycles embody the program state. The end of a cycle is synonym of completion of a transition.

Darvas et al. [11], [12], as well as Adiego et al. [1], [3], [4] have contributed to the development of PLCVerif, an open-source tool for formal verification of PLC programs using model checking, with several tools such as NuSMV [10], Theta [27] and CBMC [17]. The approach developed focuses initially on ST program formalization. IL and textual LD are also tackled. The underlying formalization approach is **intra-cycle**: it involves modeling each line of code as a state in a target transition system. Program variables are assigned along the transitions

Pavlovic et al. [22] propose to model check IL programs using NuSMV. The formalization is achieved according to an *intra-cycle* approach. It produces state variables from software-specific variables such as Boolean and integer variables, hardware-specific variables including CPU registers such as accumulators and stacks, and program counters. The initial state is defined as the initial valuation for all variables, whether they are software or hardware variables, as well as program counters. Finally, a transition relation models the evolution of all the states.

They also propose a verification approach for FBD programs [21]. Since the FBD language is a restricted form of graphical representation of the low-level IL language, it can be represented textually. FBD programs, are translated into a textual format called textFBD, which can be fed to NuSMV. Connections between the graphical FBD elements are represented in the textual file by special variables called "circuit variables". To avoid excessive use of circuit variables, which significantly increase the state space, the textual format is optimized by omitting them and producing tFBD programs, which are finally fed to NuSMV for verification.

Rawlings et al. [25], propose a formal modeling approach for hybrid systems, featuring both continuous time and discrete time behaviors. The discrete behavior is verified using NuSMV. The formalization focuses on ST programs. States are represented as vectors, where each vector component corresponds to a program variable. Transitions between states are modeled as state variables evolutions and are determined by program assignments and conditions. Data dependencies within a PLC cycle are handled, which means that not all program variables are modeled as state variables, making this approach *inter-cycle*. The authors also advocate the use of a labeled finite transition system (LTS) as a user-friendly design tool. LTSs are designed using ST. These methods are implemented in one tool: st2smv for translating ST programs into a formal model. Combinational explosion is handled using both the native heuristics of the model checker and by manually constructing model abstractions prior to the verification.

Zhou et al. [28], adopt a mapping method to model a PLC program written in ST. They establish a correspondence between the syntax of the PLC program and that of CSP#, which then serves as the entry point to the language-specific model checker. The translation follows syntactic rules and produces CSP# code in a denotational manner. They also enable verification of other languages using ST as an intermediate (pivot) language.

Niang et al. [20], translate programs written in LD and SFC into algebraic equations. These equations are represented using networks of timed automata fed to the UPPAAL model checker.

Similarly, Lampérière-Couffin et al. [18], conduct an analysis on the most common approaches to formalize a program developed in either LD or SFC. Their analysis reveals that the behavior of LD programs can be represented either by an intermediate step, translated into SFC or Petri nets before formalization, or directly modeled using temporal logic formulae.

Smet et al. [26] formally describe the behavior of PLC programs for different PLC languages. SFC programs are modeled using transition systems. Each transition is triggered by specific events or conditions and may result in a change in program variables. LD programs are formalized from a textual representation. The program behavior is described following an *intra-cycle* approach, using a transition system. Transitions may correspond to evaluating an instruction or modifying the values of variables in the program. ST programs are formalized following a FSM-based approach. Each element of the ST program, such as value assignments to a variable, conditions or loops are described using a simple automaton. These simple automata are then assembled into a global representation. The result is reorganized in order to form action sequences. Finally, composition and reduction rules are applied to build the transition system. These rules allow extending the automata and combining them to represent the complete program behavior which is fed to the Cadence SMV tool.

Rausch and Krogh [24] propose an *intra-cycle* formalization method, modeling each PLC cycle as a sequence of states.

Gourcuff et al. [13] propose an improved formalization technique, based on an advanced analysis of ST code. Variables are identified as either relevant or non-relevant through a process identifying static and temporal dependencies in the ST code, which is transformed according to an *inter-cycle* approach, so that in only contains relevant variables and their expressions which only depend on both input and relevant variables. Output assignments also depend exclusively on input variables and relevant variables. The size of the resulting NuSMV code is drastically reduced in comparison with other approaches.

Adiego et al. [2] propose a complete framework for verifying industrial-sized PLC programs. ST and SFC programs are both supported and translated to either nuXMV or UPPAAL. SFC programs are assumed to represent finite state automata (without the parallel branches), which is why they are formalized in a straightforward manner. ST (alias STL) programs are formalized according to a set of syntactic rules assuming that all variables are included in the global resulting FSM model.

C. Discussion

A synthesis of the various research efforts studied above can be done according to three axes.

1) Language support: Most research works examined are mainly focused on ST and LD, followed by SFC. It is noteworthy that the most used programming language in the studies above is ST. One probable reason is that textual languages such as ST are quite natural and easy to analyze, as highlighted by Ovatman et al. [32]. IL is supported only marginally. This restricted support is due to the deprecation of IL, on one hand, but also to a growing focus towards certified safety-critical PLCs. Certifications require, besides special hardware, a remastered design process which excludes error-prone PLC languages, with a drastic limitation to LD for certain vendors.

While LD natively provides a safe design framework, such a limitation seems excessive. Indeed, for the sake of design flexibility and most of all providing the adequate insight on a program, ST and SFC are complementary to LD. Typically, LTS models are very useful design artifacts, hard to model using LD. A promising compromise would advocate the adoption of language subsets which exclude error-prone constructs. For instance, exclude jumps and potentially unbounded constructs such as the *while* loops from ST. This is equally beneficial for applying model checking: most *inter-cycle* FSM formalization approaches presented above require such restrictions.

2) Verification performance: Model checking performance is strongly impacted by the internal FSM representation of the program at hand. There are ways to improve this performance, but still, the model checking technique does not scale well in general. Still, one way to enhance its performance is the reduction of the number of state variables. To address the combinatorial explosion problem, most studies resort to abstraction techniques, which are effective in reducing the state space. However, this approach may overlook crucial data, which can lead to false negatives.

In this context, *intra-cycle* formalization approaches are penalized: the longer the ST program, the more states in the CFG (Control Flow Graph), and the more complex the switching logic obtained. In contrast, inter-cycle formalization approaches tend to limit the number of state variables, and they are not sensitive to the number of program lines.

3) Representation Accuracy: Despite its performance drawback, *intra-cycle* formalization provides a surgical insight on complex code, possibly featuring jumps and any kind of loops. It is able to verify interesting properties related to the execution of one cycle such as loop termination. When such constructs are deemed unavoidable, *intra-cycle* formalization remains the best option. Its performance drawbacks can be mitigated by the accessibility to additional tools, such as SAT-based model checking and static analysis. These program features cannot be represented using an *inter-cycle* formalization approach, which requires by construction their absence from the program.

Based on these findings, our work advocates an *inter-cycle* approach that enhances the scalability of the model checking process, while defining an appropriate and compact PLC formal model.

III. Inter-cycle PLC program formalization approach

The inter-cycle approach relies on a set of restrictive language assumptions : jump instructions and unbounded loops are not allowed. For simplicity, only Boolean data types are considered. Floating-point types are outside the scope of this work.

A. Modeling concept

The formalization process generates a set of SSA (Single Static Assignements) [9], creating explicit dependencies between variable instances. Each variable is indexed with an integer "version" number k that increments upon reassignment. For example:

$A := B\ xor\ C$; transforms into $A\_1 \leftarrow B\_0\ xor\ C\_0$;
$A := A\ and\ D$; $A\_2 \leftarrow A\_1\ and\ D\_0$;

An assignment between a variable var and an expression $expr$ yields a new instance $var\_k ::= expr$. Expressions are represented as syntax trees which reference the appropriate k indexed instance:

$$\text{expr} ::= \begin{array}{|l} constant \in \mathbb{B} \\ (var, k) \text{ s.t. } v \in X \cup V \wedge k \geq 0 \\ expr\ op\ expr \\ conditional\_expr \end{array}$$

where op is a Boolean operator and conditional expressions are defined as:

$$conditional\_expr ::= (expr)?expr : expr$$

Hence, an ST program formalization ST_PROG is represented as a set of uniquely assigned variables :

$PROG = \{(v, k, expr) \mid v \in (S \cup O \cup X); k \geq 0$

To track the most recent assignment index k of variable v, we define:

$imax_{PROG} : (S \cup O \cup X) \rightarrow \mathbb{N}$ as

$imax_{PROG}(v) = \max\{k \mid (v, k, e) \in PROG\}$

ST code formalization is achieved according to the following rules:

1) Formalize assignments: A variable assignment $v := expr$ is transformed in:

$PROG = PROG \cup (v, imax_{PROG}(v) + 1, expr)$

For example:

Input: $A := B\ xor\ C$;

Output: $PROG = PROG \cup (A, 1, (B\_0\ xor\ C\_0))$

2) Formalize conditional statements: For an "if condition then...else..." type statement, the transformation steps are:

1) Transform **THEN** branch:
   - $PROG_T \leftarrow PROG$
   - $PROG_T \leftarrow PROG_T \cup$ THEN statements
2) Transform **ELSE** branch:
   - $PROG_E \leftarrow PROG$
   - $PROG_E \leftarrow PROG_E \cup$ ELSE statements
3) Fusion $PROG_T\ and\ PROG_E$:
   $PROG_F = \{(\mathbf{v}, \mathbf{idx_F}, \mathbf{cond\_expr}) \mid$
   $\mathbf{v} \in (PROG_T \setminus PROG) \cup (PROG_E \setminus PROG)\}$

where:

$\mathbf{idx_F} = max(imax_{PROG_T}(\mathbf{v}), imax_{PROG_E}(\mathbf{v})) + 1$

$\mathbf{cond\_exp} = (condition)?v_{idxT} : v_{idxF}$,

$idxT = imax_{PROG_T}(v)$ and $idxF = imax_{PROG_F}(v)$

For example:

Input:	**Output:**
$IF\ A\ THEN$	
$X := 1;$	$PROG_T = \{(X, 1, 1)\}$
$ELSE$	
$X := 0;$	$PROG_E = \{(X, 2, 0)\}$
$END\ IF;$	$PROG_F = \{(X, 3, (A\_0?X\_1 : X\_2))\}$

B. Building the formal model

Building the Mealy FSM reuses the NuSMV formalism. It is achieved according to four steps, detailed below.

1) *Generate all unique assignments contained by Prog*
   Each unique assignment $(v, k, e) \in PROG$ yields a NuSMV DEFINE statement:
   $$"DEFINE\ v\_k := e"$$
2) *Identify the state variables*
   This is achieved according to a simple principle: any unique assignment instance $(v, 0), v \notin X$ referenced in any program expression is considered a state variable. This amounts to targeting variables not yet assigned in the current PLC cycle.
3) *Generate δ*
   For the state variables identified above, two NuSMV ASSIGN statements are generated:
   $$S_0^{v\_0} : "ASSIGN\ init(v\_0) := 0;"$$
   $$\delta_{v\_0} : "ASSIGN\ next(v\_0) := v\_max;"$$
   where $max : imax_{PROG}(v)$.
4) *Generate λ*
   For all unique assignment instances $(v, k, e) \in Prog$ such that $v \in O$, a DEFINE NuSMV statement is generated: λ_v : $"DEFINE\ v := v\_max;"$ where $max : imax_{PROG}(v)$.

This transformation process is illustrated in the next section.

C. Illustration

Figure 2 illustrates a Structured Text (ST) program with conditional expressions, assignments and state variables. The association of unique assignment instances is visually represented, line by line. Note that, based on the unique instances obtained, the variables bool4_0 and int1_0 are explicitely referenced. Hence, they should also be considered as state variables in the final model.

IV. Performance Analysis

This section evaluates the performance of inter-cycle and intra-cycle formalizations. The analysis aims to determine efficiency trade-offs between these approaches. Based on the example illustrated in fig. 2, six other benchmarks are be generated:

- **B0 (Baseline Benchmark)**: This benchmark represents a simple program with basic assignments. It establishes a baseline for analyzing model complexity with minimal state variables and transitions.

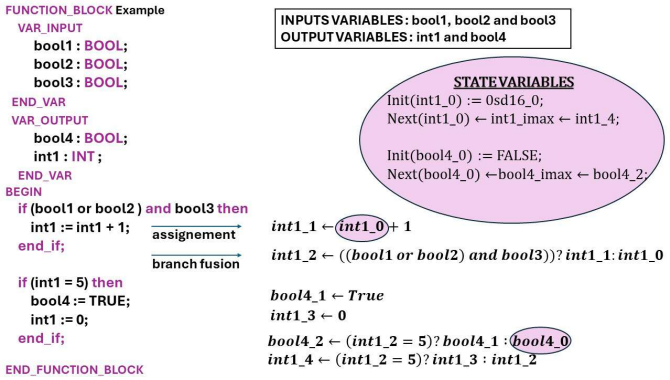

Fig. 2. ST modeling example

- **B1 (Expanded Logic)**: Includes additional lines of structured text code introducing conditional logic. This benchmark evaluates the impact of logical branching on state variable growth.
- **B2 (Intermediate Complexity)**: Features a moderate level of nesting in conditional statements. This benchmark introduces dependencies between variables, increasing the state transition complexity.
- **B3 (High Complexity)**: A significant increase in nested conditions and the introduction of modulo operations. This benchmark challenges the model's ability to manage conditional dependencies efficiently.
- **B4–B6 (Advanced Complexity)**: These benchmarks simulate real-world program complexity, including multiple interdependent variables and complex transition logic. They test the scalability and robustness of inter-cycle and intra-cycle models.

Benchmarks B0–B6 progressively increase in complexity, reflecting various real-world structured text scenarios. These allow a detailed performance evaluation across metrics such as state variable count, system diameter, memory usage and cluster size. Each benchmark is modeled using the two approaches: inter-cycle (based on the concept in Section III-A) and intra-cycle. Table I shows the performance figures obtained for inter and intra-cycle representations of the same programs. These figures are obtained using NuSMV.

A first observation reveals that for all programs, regardless of their size, the *inter-cycle* model requires fewer state variables than the *intra-cycle* model. The lower number of state variables for the *inter-cycle* representation can be explained by the fact that the *intra-cycle* model considers all assignment variables as state variables, which mechanically increases their number and the impact on the FSM model representation size, impacting ultimately the memory usage.

The system diameter is also an interesting figure: it represents the number of fixed-point iterations needed to achieve an exhaustive model exploration. Knowing that memory peaks occur during the computation of each fixed-point iteration, the larger their number, the higher the risk of verification aborting due to insufficient memory. This issue is particularly pronounced in intra-cycle models, which tend to require more fixed-point iterations, making them less appealing for efficient verification.

While cluster sizes vary between inter-cycle and intra-cycle models, depending on transition complexity and state variables, the memory footprint ultimately dominates. Intra-cycle models generally consume more memory due to larger clusters, though inter-cycle can surpass them in specific cases. Nested conditionals and added state variables seem to significantly impact cluster growth, as seen in Benchmark B3, where inter-cycle had fewer state variables yet larger clusters with lower memory use. This aspect requires further investigation. However, in the end, memory consumption remains the decisive factor in state space exploration efficiency.

Thus, inter-cycle representations provide overall better efficiency, offering a more scalable and memory-efficient approach to symbolic model checking, reducing state variable overhead, limiting fixed-point iterations, and ultimately ensuring a more practical FSM representation.

However, these trends need to be put into a broader perspective. Intra-cycle modeling remains instrumental for many PLC ST programs featuring constructs which cannot be handled by the inter-cycle approach. More specifically, loop termination analysis requires an intra-cycle representation, or a total paradigm switch towards abstract interpretation. Hence, rather than settling on inter-cycle modeling, the results presented in this work invite designer to favor as much as possible a programming style compatible with an inter-cycle modeling, in order to get a better scalability, but keep in mind the situations where intra-cycle models are required. It is also important to emphasize that due to a different interpretation of the stat/transition dynamics, intra-cycle modeling calls for a specific implementation method, totally different from the inter-cycle approach. These aspect are out of scope of this paper.

V. Conclusion

Abundant research on the applicability of model checking in PLC systems witness the significance of the field, highlighting a diverse range of formal modeling approaches, each with its own strengths, limitations, and trade-offs in verification efficiency and scalability. This work advocates a novel approach aiming to simplify the formal representation of PLC programs in order to enhance the model checking scalability. By leveraging the Single Static Assignment (SSA) technique, variable assignments are systematically tracked to construct a set of functions representing the program's evolution. This approach tackles the formalization of a sequential program, a non-trivial task that requires accurately capturing execution dependencies while ensuring a faithful transformation of the original control logic. Performance figures provided show promising results.

While the proposed approach significantly simplifies model complexity, it must be considered within a broader perspective, recognizing that intra-cycle representation remains valuable in certain contexts. Situations requiring fine-grained control flow analysis or cycle-specific properties may still benefit

	B0		B1		B2		B3		B4		B5		B6	
Number of lines	7		15		18		24		27		40		44	
	Inter	***Intra***	***Inter***	***Intra***	***Inter***	***Intra***	***Inter***	***Intra***	***Inter***	***Intra***	***Inter***	***Intra***	***Inter***	***Intra***
System diameter	10	50	3	46	43691	524282	49166	688308	43691	720982	7	146	27307	415058
Nb states var	2	6	2	6	4	8	4	8	4	8	5	9	2	6
Number of bits needed	20	23	20	24	37	42	37	42	37	42	53	59	20	26
Memory used (Mo)	4,81	4,86	4,88	4,89	31,8	88,61	62,99	284,06	31,92	108,89	5,73	6,06	23,85	66,029
Clusters size	193	260	348	403	839	1567 11	2309 3167 364	1615 544	1325	1996 31	1353 483	1975 1137 124	779	944
Nb of BDD variables	38	47	38	49	72	85	72	85	72	85	104	119	38	55

TABLE I
PERFORMANCE ANALYSIS RESULTS

from intra-cycle modeling, complementing the strengths of the inter-cycle approach. Future research will explore SAT-based verification, as well as methodological aspects for the verification of industrial PLC programs featuring intra and inter-cycle representaions.

REFERENCES

[1] Borja Fernandez Adiego, Daniel Darvas, Jean-Charles Tournier, Enrique Blanco Viñuela, and Victor M. Gonzalez Suarez. Bringing Automated Model Checking to PLC Program Development. May 2014.

[2] Borja Fernandez Adiego, Daniel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Victor M. Gonzalez Suarez. Applying model checking to industriel-sized PLC programs. *December 2015 IEEE Transactions on Industrial Informatics*, 2015.

[3] Borja Fernandez Adiego, Enrique Blanco Viñuela, Frederic Havart, Tomasz Ladzinski, Ignacio D. Lopez-Miguel, and Jean-Charles Tournier. Applying Model Checking to Highly-Configurable Safety Critical Software: The SPS-PPS PLC Program. *Proceedings of the 18th International Conference on Accelerator and Large Experimental Physics Control Systems*, ICALEPCS2021, 2022.

[4] Borja Fernandez Adiego, Enrique Blanco Viñuela, Jean-Charles Tournier, and Victor M Gonzãlez. Model-based automated testing of critical PLC programs. *Industrial Informatics (INDIN), 2013 11th IEEE International Conference*, 2013.

[5] Florian Angerer and Rudolf Ramler. Static Code Analysis of IEC 61131-3 Programs : Comprehensive Tool Support and Experiences from Large-Scale Industrial Application. August 2016.

[6] Sebastian Biallas, Kowalewski Stefan, Stattelmann Stefan, and Schlich Bastian. Efficient Handling of States in Abstract Interpretation of Industrial Programmable Logic Controller Code. *2th IFAC/IEEE Workshop on Discrete Event Systems Cachan, France.*, May 2014.

[7] Dimitri Bohlender and Stefan Kowalewski. Compositional Verification of PLC Software using Horn Clauses and Mode Abstraction. *IFAC-PapersOnLine*, 51(7):428–433, 2018.

[8] Sébastian Bornot, Ralf Huuck, Ben Luckoschus, and Yassine Lakhnech. Utilising Static Analysis for programmable Logic Controller. 2000.

[9] Marc M. Brandis and Hanspeter Mössenböck. Single Pass Generation of Static Single Assignment Form for Structured Languages. *ACM Transactions on Programming Languages and Systems ·*, page 16, November 1994.

[10] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 359–364, Berlin, Heidelberg, 2002. Springer-Verlag.

[11] DÃ¡Niel Darvas, Enrique Blanco Viñuela, and IstvÃ¡N Majzik. What is Special About PLC Software Model Checking? *Proceedings of the 16th Int. Conf. on Accelerator and Large Experimental Control Systems*, ICALEPCS2017:6 pages, 0.269 MB, 2018. Artwork Size: 6 pages, 0.269 MB ISBN: 9783954501939 Medium: PDF Publisher: [object Object].

[12] Dániel Darvas, Enrique Blanco Vinuela, and Borja Fernández Adiego. PLCverif: A Tool to Verify PLC Programs Based on Model Checking Techniques. *Proceedings of the 15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems*, ICALEPCS2015:4 pages, 0.407 MB, 2015. Artwork Size: 4 pages, 0.407 MB ISBN: 9783954501489 Medium: PDF Publisher: JACoW, Geneva, Switzerland.

[13] V. Gourcuff, O. De Smet, and J.-M. Faure. Improving large-sized PLC programs verification using abstractions. *IFAC Proceedings Volumes*, 41(2):5101–5106, 2008.

[14] Christian Huber. *A generic approach for static code analysis of PLC programs*. September 2016.

[15] Hussama I. Ismail, Iury V. Bessa, Lucas C. Cordeiro, Eddie B. de Lima Filho, and João E. Chaves Filho. DSVerifier: A Bounded Model Checking Tool for Digital Systems. In Bernd Fischer and Jaco Geldenhuys, editors, *Model Checking Software*, volume 9232, pages 126–131. Springer International Publishing, Cham, 2015. Series Title: Lecture Notes in Computer Science.

[16] Bennet Kahn. Formal verification of structures text PLC code using COQ. HONORS THESIS SUBMITTED ON THE FIFTH DAY OF MAY,2023, Tulane University, May 2023.

[17] Daniel Kroening, Peter Schrammel, and Michael Tautschnig. CBMC: The C Bounded Model Checker. 2023. Publisher: [object Object] Version Number: 1.

[18] S. Lamperiere-Couffin, O. Rossi, J.-M. Roussel, and J.-J. Lesage. Formal validation of PLC programs: A survey. In *1999 European Control Conference (ECC)*, pages 2170–2175, Karlsruhe, August 1999. IEEE.

[19] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, September 1955.

[20] M. Niang, B. Riera, A. Philippot, J. Zaytoon, F. Gellot, and R. Coupat. A methodology for automatic generation, formal verification and implementation of safe PLC programs for power supply equipment of the electric lines of railway control systems. *Computers in Industry*, 123:103328, December 2020.

[21] Olivera Pavlovic and Hans-Dieter Ehrich. Model Checking PLC Software Written in Function Block Diagram. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 439–448, Paris, France, 2010. IEEE.

[22] Olivera Pavlovic, Ralf Pinger, and Kollmann Maik. Automated Formal Verification of PLC Programs Written in IL. *VERIFY*, 2007.

[23] Herbert Prahofer, Florian Angerer, Rudolf Ramler, and Friedrich Grillenberger. Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application. *IEEE Transactions on Industrial Informatics*, 13(1):37–47, February 2017.

[24] M. Rausch and B.H. Krogh. Formal verification of PLC programs. In *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207)*, pages 234–238 vol.1, Philadelphia, PA, USA, 1998. IEEE.

[25] Blake C. Rawlings, John M. Wassick, and B. Erik Ydstie. Application of formal verification and falsification to large-scale chemical plant automation systems. *Computers & Chemical Engineering*, 114:211–220, June 2018.

[26] O De Smet, S Couffin, O Rossi, and G Canet. Safe programming of PLC using formal verification methods. November 2000.

[27] Tamas Toth, Akos Hajdu, Andras Vorcos, Zoltan Micskei, and Istvan Majzik. Theta: A framework for abstraction refinement-based model checking. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 176–179, Vienna, October 2017. IEEE.

[28] Qibin Zhou, Fangda Cai, Yang Yang, and Changshun Wu. Formal Verification of ST Programs using CSP. *Journal of Physics: Conference Series*, 2010(1):012072, September 2021.