WHAT IS SPECIAL ABOUT PLC SOFTWARE MODEL CHECKING?  
(EXTENDED VERSION)

D. Darvas*, E. Blanco Viñuela, CERN, Geneva, Switzerland 
I. Majzik, Budapest University of Technology and Economics (BME), Budapest, Hungary

Abstract

Model checking is a formal verification technique to check given properties of models, designs or programs with mathematical precision. Due to its high knowledge and resource demand, the use of model checking is restricted mainly to core parts of highly critical systems. However, we and many other authors have argued that automated model checking of PLC programs is feasible and beneficial in practice. In this paper we aim to explain why model checking is applicable to PLC programs even though its use for software in general is too difficult. We present an overview of the particularities of PLC programs which influence the feasibility and complexity of their model checking. Furthermore, we list the main challenges in this domain and the solutions proposed in previous works.

INTRODUCTION AND MOTIVATION

The promise of model checking is to provide precise, mathematically sound means to check the satisfaction of given requirements on models, representing for example software. Although some tools are available (e.g. CBMC\(^1\), BLAST\(^2\), Bandera\(^3\), DIVINE\(^4\) [3]), it is still difficult to use model checking on real-sized software in practice. One of the bottlenecks is the verification performance, the excessive need of resources for the successful verification.

Besides checking software written in general-purpose programming languages (e.g. C, C++, Java), active research can be observed focusing on PLC (Programmable Logic Controller) programs specifically. It has been studied by dozens of research groups over the last 20 years \([4]\). However, model checking is still far away from being easy-to-use or part of the state of the practice of PLC program development.

The reader may ask the question: what is the reason for targeting PLC model checking specifically? What makes this domain special and why there is a need for specific tools? What makes PLC model checking different from verifying general-purpose programs? This paper is dedicated to the particularities of PLC programs, which facilitate their verification, or contrarily, make the model checking more difficult. Our aim is to summarise our experience with PLC software model checking that we have acquired during the development of PLCVerif \([5]\), and to help formal verification researchers to specialise in this field, or to make their model checker tools applicable to the PLC program verification domain too.

The paper first overviews the difficulties and advantages arising from the domain specificities. Then the syntactic and semantic particularities of PLC programs are discussed. Finally, the need for environment modelling is mentioned. This paper is an extended version of the conference paper \([6]\).

DOMAIN SPECIFICITIES

Many of the differences between the general-purpose and PLC programming languages, but also between the available verification methods originate from the differences in the respective domains. Therefore, we start by overviewing the most important properties and particularities of the PLC domain which influence the formal verification of PLC programs.

“Medium criticality” Except for trivial programs, it is difficult to imagine and prove absolute correctness or safety, just as absolute security. Instead of pursuing those ideals, a more pragmatic approach is needed: the verification costs and the risks of failure should be in balance. Formal verification is already often used where the cost of failure is exceptionally high: in case of highly critical systems (e.g. nuclear, railway or avionics systems \([7–9]\)) or systems produced in high quantities (e.g. microprocessors \([10,11]\)). Even the methods requiring special knowledge and lots of resources may be affordable in those cases. Contrarily, in case of systems with low criticality, deep analysis may not be required.

PLC systems are in the middle of this criticality scale: their criticality is often not high enough to afford an independent, specially skilled verification team. However, a potential failure or outage may cause significant economic losses, motivating a sound and detailed verification approach.

Consequence. PLC model checking approaches should be easily accessible, specifically targeting the PLC domain, without requiring unaffordable resources or having an excessive cost compared to the level of criticality.

Advantage: Simple operations and data structure In general, the functionality of PLC programs is simpler than most programs written in C or Java. PLC programs do not deal with graphical interfaces, large data structures; they do not create files, do not perform complex operations. All these features may complicate the software model checking.

Consequence. The simplicity of the programs makes model checking more feasible computationally. This makes the PLC domain a good target for formal verification.

\(^*\) Corresponding author. E-mail: ddarvas@cern.ch  
\(^1\) http://www.cprover.org/cbmc/  
\(^2\) http://cseweb.ucsd.edu/~rjhala/blast.html  
\(^3\) http://bandera.projects.cs.ksu.edu/  
\(^4\) http://divine.fi.muni.cz/
**SYNTAX OF PLC LANGUAGES**

As mentioned earlier, PLC programs are written using a wide variety of programming languages. Since—according to their claims—Siemens is a market leader in the field of automation, we mainly focus on the languages supported by Siemens S7 PLCs, especially the high-level Structured Control Language (SCL), which is a variant of the Structured Text (ST) language defined in IEC 61131 [12].

In this section, we show that although the PLC programs are simpler, their syntax may actually be more complex than that of general-purpose programs.

**Difficulty: Variety of PLC languages** PLCs use special languages, not used outside this domain. Furthermore, there is a wide variety of PLC programming languages. IEC 61131, the relevant standard [12] defines five different languages: Structured Text (ST), Instruction List (IL), Function Block Diagram (FBD), Ladder Diagram (LD) and Sequential Function Chart (SFC). Furthermore, each vendor provides their own flavour, with minor or major differences compared to the standard. Siemens PLCs typically support Structured Control Language (SCL), Statement List (STL), Function Block Diagram (FBD), Ladder Logic (LAD), and S7-GRAPH, which correspond to the previously listed standard languages, respectively. See Figure 1 for illustration. The difference between some of them is minor (e.g. between LAD and LD), but in other cases it is very significant (e.g. between STL and IL or SCL and ST).

**Consequence.** As PLC programs can mix these languages (e.g. a function written in FBD can call a function written in SCL), each language should be supported by a PLC program verification tool. Furthermore, as there are common parts in those languages (e.g. variable declarations), the language infrastructure of the verification tool (parser and program representation) should be generic and reusable.

**Difficulty: Different background knowledge of developers** General purpose programming languages, their development environments and verification tools are typically developed “inside the community”: by software engineers, for software engineers. PLC programs, however, are often written by people with different skills and background knowledge: automation engineers, technicians, etc. The theory and practice of formal verification is often not part of the general curriculum of software engineers, making the application of model checking hard. This knowledge gap is even bigger and more severe in case of the PLC program developers.

**Consequence.** Special attention should be paid to bridge the semantic gap between the user and the verification tool. The tools should use inputs and outputs which are close to the users’ domain. For example, the PLCVerif tool [5] uses the PLC programs and requirement patterns based on English sentences as inputs, and the outputs are provided in an easy-to-understand, self-contained form, using concepts directly from the PLC domain.

**Difficulties:**

- **Complex syntax** PLC programming languages—especially their Siemens variants—often have richer and more complex syntax than general-purpose programming languages supported by software model checkers. For example, C (the C99 version) contains 6 basic data types with built-in support5. Java contains 9, but SCL contains 16 base types, which was extended to 30 in the new version of the language supported by the new development environment (TIA Portal) and the new hardware (e.g. S7-1500).

Here is the full list of the data types with dedicated support built into the languages mentioned before:

- C (C99): char, short, int, long, long, string (char[1])
- Java: char, byte, short, int, long, float, double, boolean, String
- SCL (v5.3): bool, byte, word, dword, char, int, dint, real, time, s5time, date, time_of_day, date_and_time, string, timer, counter
- SCL (TIA Portal): bool, byte, word, dword, lword, sint, usint, int, uint, dint, luint, int, real, lreal, s5time, time, ltime, date, time_of_day, ltimestamp, date_and_time, ldt, dtl, char, wchar, string, wstring, timer, counter

**Consequence.** Development of the language infrastructure for PLC software model checking needs a lot of effort. As there is no good, reusable language infrastructure available, the entry cost of PLC program verification is high.

**Difficulty: No precise syntax definition** The well-established general-purpose languages typically have precisely, often formally defined syntax. For example, the syntax of C is standardized by ANSI, ISO and IEC (ISO/IEC 9899). C# is defined by the ECMA-334 standard.

---

5 Here we not only consider the basic numeric types of C, but also strings. Even though a C string is simply a character array, there is dedicated language-level support for string constants (e.g. “var = “teststring”;

---

**Figure 1: Example of Siemens PLC languages**

<table>
<thead>
<tr>
<th>ST</th>
<th>LD (Siemens)</th>
<th>FBD</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF NOT(x = TRUE OR y = FALSE) THEN</td>
<td>A x</td>
<td>x</td>
</tr>
<tr>
<td>END_IF</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>s2 := (a &gt;= b);</td>
<td>L a b</td>
<td>lword</td>
</tr>
<tr>
<td></td>
<td>s2</td>
<td>s5time</td>
</tr>
</tbody>
</table>

---

5. Here we not only consider the basic numeric types of C, but also strings. Even though a C string is simply a character array, there is dedicated language-level support for string constants (e.g. "var = "teststring").
The Java syntax is not standard, but a detailed specification is provided by Oracle [13]. The syntax of standard PLC programming languages are defined in IEC 61131 (with some ambiguities [14, 15]). However, having a precise definition for the vendors’ flavours is not always easy. Siemens provided syntax definition for SCL version 5.3 [16] and for STL [17], which cover most, but not all aspects. The authors are not aware of any precise syntax description of the new version of SCL, supported by the new development environment, TIA Portal.

Example. The SCL v5.3 definition [16] is vague about return values. According to Section 6.13, for return values “all data types are permitted except for STRUCT and ARRAY”. Are user-defined types (UDTs), which are essentially named STRUCTs, permitted? Experiments have shown that they are, the program in Listing 1 compiles in STEP 7 5.5.

```plaintext
Listing 1: Example for UDT function return value (SCL)

The new version of SCL (supported by TIA Portal) includes new language features, such as regions, slices, etc. These new features are mentioned in [18], however the authors are not aware of a language description similar to [16] for this new version.

Consequence. As the available syntax definitions are partial or too vague, the only way to determine the precise syntax is through systematic trials with the compilers. Creating precise descriptions for the most commonly used PLC programming languages and open source, generic parser implementations could facilitate new researchers to focus on the PLC domain and also to focus the research efforts on the verification challenges.

Difficulty: Absolute and symbolic addressing Each Siemens PLC program contains an editable symbol table, which assigns names (“symbols”) to memory locations or program units (functions, function blocks, data blocks, etc.). This allows to use symbolic addressing, i.e. using names instead of absolute addresses. However, it is possible (although considered as bad practice) to mix absolute and symbolic addresses. For example, “var1 := TRUE;”, “%var1 := TRUE;” and “M4.1 := TRUE;” can have the same meaning if there is a symbol var1 defined for the memory location M4.1.

Consequence. In order to support real PLC applications, besides supporting the five languages, the symbol tables shall be supported too. The verification tools should be able to handle the mix of absolute and symbolic addresses, or at least warn the user when an object is referred to using several names.

Difficulty: Permissive grammars An additional challenge to be faced when developing the PLC language infrastructure is the permissivity of the grammars. For example, there are at least six syntactic ways to refer to a given bit in the bit memory area: absolute access (e.g. M4.1, %MX4.1; the % and X are optional) and indexed access (e.g. M[4,1]). Furthermore, symbolic access is also possible, as mentioned before.

Consequence. The language infrastructure should have a uniform internal representation to hide these redundant details and simplify the verification task.

Example. For example, the twelve different variable assignments in Listing 2 have no difference in their semantics.

```plaintext
Listing 2: Example for permissive grammars (SCL)

The MC7 bytecode generated from the above function is in Listing 3, demonstrating the equivalence of the statements.

```plaintext
Listing 3: Code generated by the compiler from Listing 2 (MC7)

6 The quotation marks denote that var1 is a symbol, however in SCL v5.3 they can be omitted if it does not cause any confusion.
Difficulty: Context-dependent grammar Another challenge in PLC software model checking arises from the context-dependent nature of the programming languages. For example, in the STL language “A ∧ A;” is a valid statement, where the first “A” stands for “AND operation” (setting the RLO status bit to RLO ∧ A, among other operations), and the second “A” denotes a Boolean variable with name “A”. Similarly, the letter “L” may stand for an instruction (load), it can be a memory area prefix (e.g. “L 0.0” denotes the first bit in the local memory area) or a variable name. “T” can mean both an instruction (transfer) and a timer value (e.g. “T 1” is the current value of timer 1).

Consequence. These features of the language have to be taken into account when choosing the technology for the language infrastructures. For example, a parser that identifies the keywords first—independently from the context—cannot successfully parse a program written in STL due to the mentioned ambiguities, or certain workarounds are required7. It also poses a challenge to provide a single, unified parser for SCL and STL.

A complete example for the context-dependent nature of the grammar can be seen in Listing 4.

```c
FUNCTION_BLOCK FB4
  VAR_INPUT
  A : BOOL;
  END_VAR
  VAR_OUTPUT
  out : BOOL;
  END_VAR
  BEGIN
  NETWORK
  A A; // optionally: A #A;
  NOT;
  = out;
  END_FUNCTION_BLOCK
```

Listing 4: Example for context-dependence (STL)

SEMIANTICS OF PLC LANGUAGES

Not only the syntax of PLC programs is rich, their semantics (i.e. the description how the programs behave during execution) may also impose additional challenges compared to formal verification of general-purpose programs.

PLC execution semantics To provide verification for PLC programs, first the key semantic differences between general-purpose programs and PLC programs have to be understood.

PLC programs are typically executed cyclically. A cycle (so-called scan cycle or PLC cycle) consists of (1) sampling the physical inputs (and keeping their values stable in the memory), (2) executing the user code, (3) assigning the computed outputs to the physical outputs. This allows to have consistent input and output signals.

In Siemens PLCs, the scan cycle can be interrupted. Cyclic interrupts ensure the periodic execution of a certain piece of code. Diagnostic and error handling interrupts can also be defined. The interrupts and various operating system tasks (e.g. communication) can alter the length of the scan cycle. If the scan cycle exceeds the predefined length, an error-handling block will be executed.

There is a difference in the programming concepts too. Even though the latest IEC 61131 standard introduced object-oriented programming for PLCs, most programs still use functions and function blocks. A function block is a stateful function, the values of its variables (except for the temporary variables) are kept even after the execution of the block. The semantics of a function block is similar to a class that has a single member method in object-oriented languages. To illustrate the semantics for the readers new to PLC programming, a simple PLC program written in SCL is shown in Listing 5, together with its rough semantic equivalent in Java in Listing 6.

```c
FUNCTION_BLOCK FB5
  VAR_INPUT
  IN1 : INT;
  END_VAR
  VAR_OUTPUT
  OUT1 : INT;
  END_VAR
  BEGIN
  OUT1 := -1 * IN1;
  END_FUNCTION_BLOCK
```

Listing 5: Example PLC program (SCL)

Advantage: Simple memory handling The formal verification of PLC programs is greatly facilitated by its simple memory handling. PLC programs use static typing: variables are declared explicitly, with a given type. Variables are strongly typed: except for some safe cases, explicit type conversions are required between the different data types. However, it has to be noted that SCL permits the use of the special data type POINTER and ANY which can store addresses of other data. Typically, these addresses are referenced in system functions and function blocks, not in the user code. The exact structure and bit-level representation of these special data types can be found in [18, pp. 3503–3508].

There is no dynamic memory allocation in PLC programs, all variables and data blocks are allocated statically, at compile-time. It has to be noted that SCL permits the use of system functions to create new shared data blocks in run-time (CREAT_DB, SFC 22), however it only permits to address another chunk of memory and it is not fundamentally different from using memory in the M memory area. To the authors’ best knowledge it is not possible to create new instance data blocks.

Furthermore, in high-level PLC programming languages (e.g. SCL) pointers are rarely used. In lower-level languages

(e.g. STL) pointer usage is sometimes unavoidable. However, even without using pointers explicitly, semantically equivalent constructs may be present. For example, IB[10] denotes the value of byte 10 in the input memory area. If the IB array is indexed with a variable (IB[var1]), then var1 practically behaves as a pointer.

**Consequence.** Due to simple control structures and the lack of dynamic memory allocation, many popular model checkers, e.g. NuSMV or UPPAAL can efficiently be used to verify most of the PLC programs. To support all PLC programs, pointer support is required on the verification side.

**Difficulty: Imprecise semantics definition** Having a precise, formal semantics for the input models is an obvious requirement for model checking. Unfortunately, there is no mathematically sound semantics definition neither for the IEC 61131 languages, nor for the Siemens PLC languages. Some reference manuals are available for SCL [16] and STL [17], but they are in some places ambiguous, imprecise or incorrect. For example, the SCL description does not define precisely the semantics of **CASE** statements, or the STL description incompletely and sometimes incorrectly defines the behaviour of the nesting stack used for complex Boolean operations.

**Example.** The SCL documentation [16] does not define what happens if in a **CASE** statement several cases match the value of the selection expression. According to the documentation, “when a CASE statement is processed, the program checks whether the value of the selection expression is contained within a specified list of values. If a match is found, the statement component assigned to the list is executed” [16].

Based on the execution of the program in Listing 7 we can conclude that in case the value of the selection expression matches several cases, only the first matching case will be executed, and the rest will be ignored.

```java
public class Main {
    // Representation of function block FB5
    public static class FB5 implements FB {
        public int IN1;
        public int OUT1;
        public void execute() {
            OUT1 = -1 * IN1;
        }
    }
    // Representation of instance data block DB4
    public static final FB5 DB5 = new FB5();
    // Representation of function FC5
    public static int FC5() {
        DB5.IN1 = 5;
        DB5.execute();
        return DB5.OUT1;
    }
}
```

Listing 5 (Java)

Based on the execution of the program in Listing 7 we can conclude that in case the value of the selection expression matches several cases, only the first matching case will be executed, and the rest will be ignored.

```java
FUNCTION FC7 : INT
VAR
    i : INT;
    c : INT;
END_VAR
BEGIN
    i := 10;
    c := 0;
    CASE i OF
        0..20: c := c + 1;
        5,10,15: c := c + 2;
        10: c := c + 4;
        ELSE:
            c := c + 8;
        END_CASE;
    FC7 := c; // returns 1
END_FUNCTION
```

Listing 7: Test program to determine the semantics of **CASE** statements (SCL)

PLC programs depend on a library of basic functions and function blocks, such as timers, data transmission blocks, special memory operations. Precise description (either formal definition or source code) is required for these program units too for the verification, but it is often not available.

**Consequence.** Developers of PLC verification tools cannot fully rely on the provided language descriptions and documentation. Systematic, rigorous experiments have to be conducted in order to explore the precise semantics of the different PLC program structures, such as in [19].

**No short circuit evaluation** The IEC 61131 standard permits the short-circuit evaluation for logic expressions (“Boolean expressions may be evaluated only to the extent necessary to determine the resultant value.” [12, Section 3.3.1]), i.e. the evaluation can be interrupted as soon as the result can be determined. However, our experiments showed that Siemens PLC programs do not use short circuit evaluation. For example, in case of the “func1() OR func2()” expression the function func2 will be called even if the return value of func1 is true (thus the expression will be evaluated to true independently from func2).

**Consequence.** This may facilitate the representation of PLC programs as control flow graphs.

**Difficulty: Timed behaviour** PLC programs often involve time-related behaviour, typically by using the timers defined in [12] (TP, TON, TOF). Accurate modelling and verification requires precise representation of time, which might make the verification task extremely difficult. In reality, PLC timers rely on the PLC’s real time representation. The elapsed time between two timer calls depends on the cycle time, which in turn relies on the executed methods, the precise type of the hardware, the communication between the PLC and other systems, etc. More detailed discussion on
the time representation for PLC program verification can be found in [20].

Consequence. The verification tool should use an appropriate time representation, i.e. an appropriate trade-off between precision of modeling and needed resources. One possibility is to simplify the physical time handling and assuming that each PLC cycle takes a non-deterministic amount of time, and the global time is incremented by this value at the end of the cycle at once. Then effectively the time does not elapse during a PLC cycle, which may alter the behaviour of the timer blocks, but this was often found to be an acceptable trade-off. This representation may lead to false negatives, i.e. omitted faults. Other time representations could cause false positives (false error reports). The consequences of the chosen time representation shall be clearly described for the user, using the terminology of the PLC domain.

Difficulty: Semantics depending on the compiler and hardware version The precise semantics of PLC programs may depend on various compiler settings, the used compiler and the hardware.

• Certain data types and languages are available only using certain hardware. For example, the 64-bit types, such as LWORD and LREAL are only available in the newer, S7-1200 and S7-1500 series PLCs [18].

• The precise semantics of the programming languages depend on the development environment. Example. Let \( D \) be a variable of type DINT (32 bit signed integer). Using the STEP 7 V5.5 development environment, the execution of the SCL assignment “\( D := \text{INT}#1 + 50000 \)” will result in \( D = 50001 \) (where “\( \text{INT}#1 \)” denotes a 16 bit signed integer). However, the same code compiled using the TIA Portal development environment will result in \( D = -15535 \) on the same hardware due to the differences in typing rules. See Listing 8 for illustration. This behaviour is due to the differences in type conversion: the new compiler determines the result type of “\( \text{INT}#1 + 50000 \)” based on the explicit type of the constant 1, denoted by \( \text{INT} \), and the constant 50000 will also be treated as an \( \text{INT} \), which is a 16-bit signed value. Therefore, even though the number 50001 could fit into a variable of type DINT, an overflow will occur [18, pp. 3430–3432].

Example. One of the undefined parts of the SCL semantics is the evaluation of the final value (upper limit) of a for loop. See Listing 9 for illustration. It is defined that the upper limit is evaluated only once [16, Section 12.2.5] (unlike in C, where it is evaluated at the end of each execution of the loop body). However, when compiled in TIA Portal, the final value evaluation happens after the initial statement (e.g. “\( I := 1 \)” in the example) has been executed, while it is evaluated before setting the initial value if compiled with STEP 7.

Example. Consider the example SFC shown in Figure 2 that consists of three consecutive steps. The transitions between the steps are unconditional. In the Block settings panel, the user may change some “Sequencer properties”, for example, there is an option labelled “Skip steps”. The behaviour of this example SFC depends on this option.

- If the “Skip steps” option is off, the initialisation of the SFC (call with INIT_SQ=true) will activate Step1 (\( S._\text{N0}=1 \)). The next call of the SFC will make Step2 active (\( S._\text{N0}=2 \), assuming that INIT_SQ is false for this second call). For each call at most one transition can fire.

- If the “Skip steps” option is on, the initialisation of the SFC (call with INIT_SQ=true) will not activate any steps (\( S._\text{N0}=0 \)). This is because the unconditional transitions allow reaching the branch stop already at the initialisation. Then the described state machine is in a deadlock, no further transitions are permitted, therefore no further call will activate any steps.

This behaviour is documented (“Skip Steps: If both the transition before a step and the transition after the step are valid at the same time, the step does not become active and is skipped.” [21]), however the current setting is not clearly indicated in the editor, potentially misleading the developer. The current setting is represented in the textual representation of the SFC (see the option SSkipOn in the CMPSET part of Listing 10, line 11),

Listing 9: Example for semantic differences between compilers (SCL)

<table>
<thead>
<tr>
<th>FUNCTION FC9 : DINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR I : INT;</td>
</tr>
<tr>
<td>END_VAR</td>
</tr>
<tr>
<td>BEGIN</td>
</tr>
<tr>
<td>I := 5;</td>
</tr>
<tr>
<td>FOR I := 1 TO 1 DO</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>END_FOR;</td>
</tr>
<tr>
<td>FC9 := I;</td>
</tr>
<tr>
<td>END_FUNCTION</td>
</tr>
</tbody>
</table>

Listing 8: Example for semantic differences between compilers (SCL)

<table>
<thead>
<tr>
<th>FUNCTION FC8 : DINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR I : INT;</td>
</tr>
<tr>
<td>END_VAR</td>
</tr>
<tr>
<td>BEGIN</td>
</tr>
<tr>
<td>FC8 := INT#1 + 50_000;</td>
</tr>
<tr>
<td>// When compiled in TIA Portal, return value is –15535.</td>
</tr>
<tr>
<td>// When compiled in STEP 7, return value is 50001.</td>
</tr>
<tr>
<td>END_FUNCTION</td>
</tr>
</tbody>
</table>
but the corresponding parts are not documented in [21], making the verification more challenging.

- The hardware configuration and the interrupt configuration can also influence the precise semantics of a given PLC program. Furthermore, this information is not included in the source code.

Consequence. Different semantic variants of PLC languages shall be supported, and the user shall be able to choose the appropriate one for each program under verification. As programs should not depend on the unspecified semantics of the language, in such cases it may be sufficient to identify the reliance on undefined features instead of precisely handling the semantic alternatives.

Difficulty: Bit-level memory manipulation

PLC programs allow various low-level memory manipulations.

- Integer (and even floating-point) variables can also be treated as bit arrays by using explicit type conversion operators (see Listing 11 for example). The same behaviour is also possible by defining so-called views, practically declaring multiple variables mapped to the same memory location using the keyword AT in SCL. In other words, the raw data stored at a certain memory location can be interpreted in different ways through the different views. This can be helpful for example when a WORD needs to be accessed bit-by-bit (as an array of BOOLs). See Listing 12 for example. Note that the indices of the view may be misleading or unintuitive, see Listing 13.

- It is also possible to directly address a specific area in the memory (absolute addressing), independently from the variable borders. For example, DB1.DW3 refers to the WORD starting at byte 2 in the data block DB1. However, this memory location may represent several variables, or parts of different variables. See Listing 14 for illustration.

- When a certain part of the memory is accessed both using its address and symbolically (i.e. using its name), it is obvious that the user needs to be aware of the memory layout that the compiler creates. However, this

Listing 10: Textual representation of the example SFC

Listing 11: Bit manipulation on numbers (SCL)

Listing 12: Bit manipulation using views (SCL)
Listing 13: Bit manipulation using views with non-zero-based indexing (SCL)

Listing 14: Bit manipulation by absolute data block addressing (SCL)

Listing 15: Variable views affected by physical memory layout (SCL)

Figure 3: Memory layout of the program in Listing 15

ENVIRONMENT

Challenge: Environment model  PLCs are mainly used for process control tasks, therefore they inherently interact with their environment. It is reasonable to check certain safety properties (i.e. a given property is always satisfied, no matter what are the input sequences) without considering the environment during verification. However for other types of requirements having no assumption on the environment may lead to many false positives, i.e. non-satisfied requirements where the violation is practically impossible.

Consequence. To get practical, usable verification results, the model of the environment needs to be incorporated. This can exclude cases where for example only a physically impossible change in the controlled process could cause the signalled violation. In our opinion, there are three main challenges related to the environment models, as follows.

• It is difficult to find appropriate formalisms and to describe the environment (e.g. the controlled process) precisely.
• Including the environment model may significantly increase the computation resources required for model checking.
An imprecise environment or process model may lead to false negative results, i.e. it can lead to the omission of real problems.

There are various attempts to precisely describe environment or process models and include them in various verification procedures [22–24], however, we think that this still remains one of the greatest challenge in PLC model checking.

**Challenge: Fault assumptions** It is important to keep in mind that the input variables of the PLC programs often represent physical inputs. It is unrealistic to assume that all inputs are always correct. In other words, the “no failure” assumption in the environment model during verification may hide potential problems. The other extreme—assuming that everything can fail at the same time—may be unrealistic too, leading to useless counterexamples which undermines the usability of the method.

**Consequence.** The environment models shall be able to incorporate various assumptions. For example, a single failure hypothesis may be rational in some cases, but in other cases including the simultaneous failure of certain dependent signals in the verification may be desired too.

**OUR RESPONSE: PLCverif**

To overcome most of these challenges and to provide feasible, easy-to-use formal verification for PLC programs, CERN started the development of PLCverif [5]. With the ongoing development of PLCverif we aim to provide a generic tool and language infrastructure that can make the development or integration of new verification methods to the PLC domain significantly easier.

PLCverif hides the formal verification-related details from the user. Also, as it relies on a control flow graph-based intermediate representation that is independent from the PLC programming languages, this tool can hide many of the syntactic and semantic peculiarities of the PLC domain from the (formal) verification solutions.

Recognizing that the listed particularities make the development of any verification method challenging for PLC programs, PLCverif is opening towards supporting other verification techniques besides model checking, for example static code analysis and unit testing.

Although we have overcome many syntactic and semantic problems—except the ones which would have required unreasonable amount of resources compared to their pertinence, such as properly supporting pointers—, the lack of proper environment modelling limit the use of PLCverif to well-defined, isolated parts of PLC applications, such as individual function blocks or safety logic implementations.

**CONCLUSION**

In this paper many of the specific challenges of model checking PLC programs have been presented, as well as the features of those programs which can facilitate their formal verification. We believe that PLC model checking is still a research field with a lot of industrial attention and with many unsolved challenges.

On one hand, PLC program verification is an ideal target for model checking due to the medium criticality and the relatively simple programs.

On the other hand, syntax and semantics of PLC programs are complex, which makes it difficult for non-PLC experts to contribute to verification, as the knowledge and development effort required for PLC program verification is high. Open source, reusable language infrastructures could leverage this challenge, allowing to focus on the challenges of performance and clarity of results. We need a bridge not only between formal verification and the PLC developer community, but also between the formal verification researchers and the industrial control systems domain. Furthermore, environment modelling is still a big challenge to be solved, which could significantly improve the practical applicability of model checking for PLC programs.
REFERENCES