PLC Program Translation for Verification Purposes

Dániel Darvas, István Majzik, and Enrique Blanco Viñuela*

Abstract

Programmable logic controllers are typically programmed in one of the five languages defined in the IEC 61131 standard. While the ability to choose the appropriate language for each program unit may be an advantage for the developers, it poses a serious challenge to verification methods. In this paper we analyse and compare these languages to show that the ST programming language can efficiently and conveniently represent all PLC languages for formal verification purposes. Furthermore, we provide a translation method from IL to ST programming languages (for the Siemens implementation), together with a sketch of proof for its correctness. This allows the usage of the ST-based PLCverif model checking method for safety PLC programs.

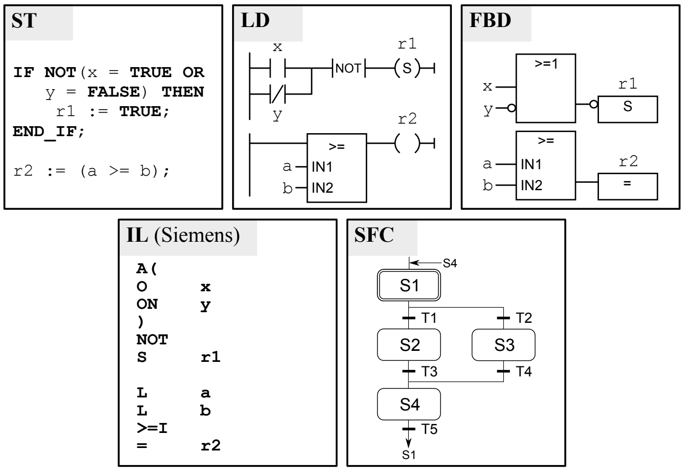
Keywords: PLC, programming languages, formal verification, semantics

1 Introduction and Background

Programmable Logic Controllers (PLCs) are widely used for various control tasks in the industry. As they often perform critical tasks – sometimes PLCs are even used in safety-critical settings up to SIL3 – the verification of these hardware-software systems is a must. Besides the common testing and simulation methods, formal verification techniques, such as model checking are used increasingly often.

The corresponding IEC 61131 standard defines five PLC-specific programming languages: Instruction List (IL), Structured Text (ST), Ladder Diagram (LD), Function Block Diagram (FBD) and Sequential Function Chart (SFC) [15]. It is out of the scope of this article to discuss the features of these languages in detail, but a simple example in Figure 1 illustrates these languages. The first four example program excerpts are *execution equivalent*, i.e. for all

*D. Darvas and I. Majzik are with Budapest University of Technology and Economics (BUTE), Budapest, Hungary. Address: H-1117 Budapest, Magyar tudósok körútja 2., Hungary.
D. Darvas and E. Blanco are with the European Organization for Nuclear Research (CERN), Geneva, Switzerland. Address: CH-1211 Geneva 23, Switzerland.



```
IF NOT(x = TRUE OR
   y = FALSE) THEN
    r1 := TRUE;
END_IF;

r2 := (a >= b);
```

```
A(
O      x
ON     y
)
NOT
S      r1

L      a
L      b
>=I
=      r2
```

Figure 1: PLC language examples

possible starting (input and retained) variable valuations, the results of these programs are the same variable valuations. The SFC example is different from the others, as this is a special-purpose language for structuring complex applications.

This variety of languages responds to the fact that PLCs are used in different settings and programmed by people with various backgrounds. This is an advantage for the developers, but an important challenge for the verification. The languages can be freely mixed, e.g. a function written in IL can call an ST function. To provide a generally applicable formal verification solution, all these languages should be supported.

1.1 Motivation

Our practical motivation lies in the PLCverif formal verification tool and its verification workflow [12, 7]. The PLCverif tool provides a way for PLC program developers to apply model checking to their implementation. This allows to check the satisfaction of various state reachability, safety and liveness requirements. The inputs of the model checking workflow are the source code and the requirements formalised using requirement patterns. At the moment, programs (or program units) written in the Siemens variant of ST are supported natively. These inputs are selected to be convenient for the users who are not familiar with formal verification methods. PLCverif automatically generates temporal logic expressions from the pattern-based requirements, parses the input code, builds a simple, automata-based intermediate verification model, calls the chosen external model checker tool (e.g. nuXmv), and presents the results in a simple, self-contained format to the user. The tool is in use at the European Organization for Nuclear Research (CERN) to check critical control programs [13]. While most of the PLC programs are written in ST at CERN, in special cases (e.g. safety instrumented systems) restrictions forbid the use of ST. To make PLCverif generally applicable, all five PLC programming languages should be supported.

From the development point of view, providing a complete parser and a verification model builder is a great effort. Furthermore, the grammars of the PLC languages are notably different, making it difficult to use the same technology stack. For example, in case of PLCverif the currently used Xtext-based[1] parser is not suitable for the IL language, where the same tokens can be treated as keywords or names depending on the context. On the other hand, the different languages have many common parts, e.g. function and function block declarations, variable declarations. If these should be developed for each language independently, the maintenance of the tool may become difficult.

Instead, in this paper we investigate the possibility of a different approach: *is it possible to use the ST language as a pivot to represent all five standard PLC languages?* If the translation preserves the properties of the model to be checked, adding this extra translation step (i.e. transformation to ST, then parse and build the verification model) makes no theoretical difference, the pivot language might be considered as a concrete syntax of the underlying intermediate verification model (IM). However, as it will be discussed later, the required development and maintenance effort could be significantly lower.

To answer this question, the relations between the PLC languages have to be investigated. As it might not be possible or practical to translate each language *directly* to ST, the relationship between all languages should be discussed.

This paper is a modified and extended version of the local conference paper [9], and it is structured as follows. Sections 2–5 are taken from the conference paper [9] with minor modifications. Section 2 defines our comparison method. Section 3 discusses the relations between the different IEC 61131 PLC programming languages. Next, Section 4 discusses a concrete implementation of these languages, namely the one provided by Siemens. Section 4.1 is an additional subsection compared to [9] that overviews the organisation of Siemens PLC programs. Section 5 analyses the results of the paper and shows that the Siemens implementation of the ST language can serve as a pivot language for the rest of the languages.

The new contributions are in Sections 6–9. Section 6 describes and formalises the IL to ST translation (using the Siemens implementation of these languages), as proposed in Section 5. Section 7 provides a method for the correctness proof of this translation. Section 8 discusses the application of the contributions of this paper for verification purposes. The reader finds an overview of the related work in Section 9. The paper is concluded in Section 10.

2 Comparison Method

The expressive power of different programming languages is often discussed in computer science. However, the typical answer to these questions for a pair of

[1] `https://eclipse.org/Xtext/`

commonly used languages is that both languages are Turing complete, therefore their expressive power is equivalent.

For our purposes this is not a useful comparison. When we are looking for a pivot language, it is not enough to know that a certain program can be represented in another language, i.e. for each program in source language S there *exists* an execution equivalent program in language T. It should be known as well, *how* can this translation be performed. Therefore we are interested in a stronger, element-wise emulation relation that determines whether *each “element”*[2] of a language S can be mapped to language T. If this relation holds, then inductively all programs of language S can be translated into language T, in other words language T can emulate language S. This is close to defining a *small-step operational semantics* for language S in language T.

In the following we investigate for each pair of PLC languages if such element-wise mapping relation exists. Note that this relation is transitive, reflexive and asymmetric. We start the investigation with the IEC 61131 version of the languages, as they have a detailed, yet semi-formal description in [15]. Later, we check the differences between the standard and the Siemens variants.

3 Standard Languages

In this section, we discuss the element-wise representation relation for each pair of standard PLC languages. The findings are summarised in Table 1. Here “−” denotes that the element-wise representation is not possible.

Table 1: Element-wise mapping between standard languages

to from	ST	IL	FBD	LD	SFC
ST	+	+	−	−	−
IL	−	+	−	−	−
FBD	−	+	+	+	−
LD	−	+	+	+	−
SFC	+	+	+	+	+

SFC is based on a specification method called Grafcet [14], which itself has roots in safety Petri nets. The goal of SFC is to structure the programs, it is not intended to be a generic PLC language. Only certain types of program units can be represented by SFCs, while the other four languages target all of the program unit types, therefore no other language can be universally represented in SFC. Since it is based on Petri nets, translating the structure of an SFC program to any other language could be problematic, because Petri nets allow non-determinism, while the PLC languages are deterministic. However, determinism is explicitly required by the standard [15, Sec. 2.6.5]. The parts of SFC besides the structure are the actions, which can can be typically regarded as

[2] As the PLC languages are significantly different, “element” is understood on a high level (i.e. an element can be an ST statement, but also an LD wire junction).

simple program snippets and these specific parts can be mapped to any other PLC language, assuming that the ambiguities of the standard are first resolved [1].

FBD and **LD** are two similar graphical languages. FBD is composed by signal flow lines and boxes representing built-in and user-defined program units. LD builds on concepts close to electric diagrams, such as power rails, contacts and coils. Despite the differences, IEC 61131 defines LD and FBD in a similar way, with many common elements. The differences [15, Sec. 4.2–4.3] are minor and mainly syntactic. All LD-specific elements (e.g. coils, power rails) can be translated to equivalent FBD elements and vice versa. The wires and flow lines represent data flows, the coils and contacts have corresponding instructions in IL (see below). The built-in and user-defined blocks of FBD and LD can be called from IL as well. Therefore each FBD and LD program can be element-wise mapped to IL, in some cases requiring to explicitly introduce new variables that are only implicitly present (as wires) in the FBD and LD programs.

Contrarily, LD and FBD programs cannot be element-wise mapped to ST. The FBD, LD and IL languages support labels and jumps, but ST enforces structured programming, thus jumps are missing from the language [15, Sec. B.3]. Although it is known that Turing complete programs can be made jump-free by replacing jumps with loops and conditional statements [6], this construction does not fit to our approach of element-wise mapping.

IL is an assembly-like, low-level language. It has instructions such as `LD` (load value to accumulator) or `ST` (store the accumulator value to the given variable). As the rest of the languages do not provide direct access to the accumulator, the element-wise (instruction by instruction) translation to any other PLC language is not possible. Furthermore, the lack of jump instruction in ST would make the IL to ST translation difficult too.

ST is a high-level, structured textual language. Besides providing program structuring elements, such as conditional statements (`IF`, `CASE`) and loops, it also makes the indirect variable access possible. For example, the expression "`array_var[var1]`" is permitted in ST, but not in FBD or LD [15, Sec. 2.4.1.2], therefore the ST to FBD or LD translation is not possible. On the other hand, these expressions are permitted in IL. More precisely, the ST syntax for defining expression is permitted in IL in certain cases [11]. Based on the syntax and semantics definitions of ST and IL, each ST statement can be represented by a list of IL instructions: the corresponding arithmetic operations exist in IL as well, the variable assignments can be performed through `LD` and `ST`, the selection and iteration statements can be represented by labels and jumps, etc.

Based on the discussion above and Table 1, ST does not seem to be a pivot language candidate. However, before the final conclusion, the *implementation* of the languages should also be checked for two reasons: (1) the different manufacturers may have differences in their implementation compared to the standard, and (2) the IEC 61131 standard is ambiguous [1, 10, 16] and the vendors might resolve the ambiguities differently. The following section compares a concrete implementation of the five PLC programming languages.

4 Implementation of the Languages

The IEC 61131 standard does not discuss the implementation details of the languages. Several decisions are left to the vendors, marked as "implementation-dependent" features or parameters in the standard (e.g. range of certain data types, output values on detected internal errors). Consequently, PLC providers support different variants of the languages. The implementation-dependent details are also important for the behaviour of the programs, thus it is necessary to check these details. Siemens is the PLC provider most used at CERN and one of the most used globally, therefore we focus on the Siemens variants of the languages in this section. All five languages are supported in Siemens PLCs with some differences [28]. Compared to the standard, the differences are significant in some cases, also some languages have ancestors from times before IEC 61131, thus Siemens uses different names for their languages: instead of ST, IL, FBD, LD, SFC the Siemens languages are called SCL, STL (AWL), FBD, LAD, SFC/GRAPH, respectively. To avoid the confusion of the readers, we will use the standard language names for the Siemens variants too, with an added apostrophe (e.g. we will use ST' instead of SCL to refer to the Siemens implementation of the standard ST language).

4.1 Organisation of Siemens PLC Programs

To make the following discussions more understandable, a brief overview of the Siemens PLC program structure is presented. This follows the IEC 61131 with some notable differences. PLCs have a mainly cyclic behaviour. In each so-called *scan cycle* (or PLC cycle) the physical inputs are read, the user-defined program is executed, then the resulting output values are assigned to the physical outputs.

The code to be executed in each cycle is described by an *organization block*. This is a special program block, similar to a function, that serves as an interface between the operating system and the user code. Similarly, the interrupt handler methods are organization blocks too. There are two other program block types: *functions* and *function blocks*. Function blocks are stateful functions: they can be instantiated and certain variables preserve their values between two calls. There are only few restrictions on the implementation of the program blocks: the SFC' language can only be used for function blocks. The other four languages can be used for all three types of program blocks. The program blocks can call each other (except organization blocks which can only be called by the operating system), no matter in which language they are written.

An additional block type exists: the *data block*. A data block is a structured memory storage. *Instance data blocks* are used to store the non-volatile variables of the function block instances, while the *shared data blocks* are used for general-purpose global storage of structured data.

It is worth to be noted that the Siemens PLC programming languages are not case sensitive.

Table 2: Element-wise mapping between Siemens languages

from \ to	ST'	IL'	FBD'	LD'	SFC'
ST'	+	+	−	−	−
IL'	−	+	−	−	−
FBD'		+	+	+	
LD'		+	+	+	
SFC'	+	+	+	+	+

4.2 Relations Between the Siemens PLC Languages

The following part of the section overviews the differences between the standard and Siemens implementations of the PLC programming languages, compared to Table 1. The differences between the standard and Siemens versions of FBD and LD are subtle and mainly syntactic[3] [28]. Notable differences in syntax and semantics between the standard and the implementation can be observed in the Siemens variants of ST and IL. The relations between the Siemens languages are shown in Table 2 and discussed in the following.

As the FBD' and LD' languages match nearly perfectly the standard versions, the relations between them are valid for the Siemens variants too. The SFC' language is an extension of the standard SFC. A notable difference is the introduction of new actions in SFC'. As SFC' has a rich and configurable semantics that is only described informally, without details, we omit the deep analysis of this language and we assume that each action can be translated into any of the languages, therefore the complete SFC' program can be translated to other languages. ST' and IL' are extended compared to the standard equivalents. Therefore if a language can be mapped to ST or IL, it can also be mapped to the corresponding implementation (ST' or IL'). Similarly, if ST or IL cannot be mapped to a given standard language, IL' or ST' cannot be mapped to the same language either. Consequently, the shaded cells of Table 2 are inherited from Table 1.

Due to the limitations of the Siemens development environment, the **FBD'** and **LD'** programs can only be exported if they are translated to IL' first. According to [22], the translation from LD' and FBD' to IL' is always possible. We omit the discussion of transforming LD' and FBD' directly to ST' or SFC', as they would be practically infeasible.

The Siemens variant of **ST** is significantly extended compared to the standard. It includes labels and jump functions, which invalidates the reasoning of Section 3 why IL, LD and FBD cannot be represented in ST. Despite the extensions, it is not possible in ST' to directly access the registers, e.g. modifying the contents of the accumulators. For example, the IL' instruction "`L var1`", transferring the contents of Accumulator 1 to Accumulator 2 and then loading the content of variable `var1` to Accumulator 1 cannot be directly represented

[3] For instance, LD' fully implements the standard. The only difference between FBD and FBD' is that the latter does not support the unconditional jumps, but it is easy to represent them as conditional jumps [28].

```
LD a  (* RES:=a *)
GE b  (* RES:=(RES>=b) *)
ST r  (* r:=RES *)
```

Listing 1: Example IL code

```
L a   (* ACC2:=ACC1; ACC1:=a *)
L b   (* ACC2:=ACC1; ACC1:=b *)
>=I   (* RLO:=(ACC2>=ACC1) *)
= r   (* r:=RLO *)
```

Listing 2: Example IL' code

in ST'. One can argue that a function containing only the instruction "`L var1`" is meaningless, as its effect will be made invisible when the function returns. However, this example is enough to demonstrate that the element-wise mapping is not possible.

The Siemens **IL'** variant is significantly different from the standard IL, and it dates back well before the first version of the IEC 1131 standard (the predecessor of IEC 61131), as this programming language was used already for the S5 series of PLCs in the 1980s. The following short example illustrates the syntactic differences. The IL program in Listing 1 and the IL' program in Listing 2 give the same output values to the same input values, but they use a significantly different syntax and underlying semantics. The behaviour of both code snippets is equivalent to "`r:=(a >= b)`" in ST. The background of this difference is that the standard defines only one "register", the result variable. The Siemens implementation is closer to the assembly-like languages, using several status bits, registers, accumulators, etc.[4] As the ST' and IL' language definitions are non-formal, it is difficult to argue about the ST' to IL' transformation. However, the Siemens development tool provides this transformation capability, therefore we treat this as possible.

5 Finding a Pivot Language

Looking at Table 2 might lead to the same conclusion for the Siemens implementations of the languages as Table 1. However, due to the extensions in the implementations, the gap between IL' and ST' is much smaller than between their standard equivalents. The only main difference between them is the possibility to access the registers directly. Therefore ST' can be a pivot language, if it is extended with the *emulation of register access* by using dedicated local variables for verification purposes. We will refer to this format of the programs as *STr'*. As the values of the registers are saved on the stack on each function call, their values are local to each program unit, they can be represented as local temporary variables. Thus the mapping from IL' to STr' can be done instruction by instruction, by explicitly representing the effects of each instruction on the basis of their semantics. For example, the above-mentioned "`L var1`" will be represented as "`ACC2 := ACC1; ACC1 := var1;`", where `ACC1` and `ACC2` are the local variables representing Accumulator 1 and 2. This idea is similar to the SystemC representation used in [29].

[4] From this point we use the term "register" in a generic way, referring to the various low-level data structures: status bits, accumulators, nesting stack, etc.

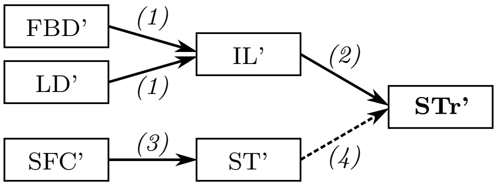


Figure 2: Unified representation of Siemens PLC languages

Although the FBD’ and LD’ programs cannot be directly translated to STr’ in practice, it is feasible through IL’. The SFC’ programs can directly be mapped to ST’, thus to STr’ also. The advantage of this method is that one parser and one intermediate verification model generator fit all the languages. Only a simpler, text-to-text mapping to STr’ has to be developed for each language that is responsible for translating the language-specific parts, element by element.

One could argue that IL’ might be a good pivot language without defining any extension or representation convention for the verification. However, STr’ is a higher-level language, with a more compact representation (especially the expression description is more compact). The underlying intermediate verification model supports also complex expressions (similarly to the formalism of many model checkers, e.g. nuXmv, UPPAAL), therefore translating a compact ST’ expression to a lengthy IL’ form is inefficient. Also, in our setting typically ST’ codes are verified, therefore using STr’ (and not IL’) as pivot can provide support for the other languages without any impact on the verification of ST’ programs.

Figure 2 summarises the proposed generic representations of PLC languages for PLCverif. The FBD’ and LD’ graphical languages can be translated into IL’ by the Siemens development environment *(1)*. An instruction-by-instruction transformation from IL’ to STr’, that makes the effects of the IL’ instructions explicit, is implemented for the most common instructions (currently we are focusing on the instructions that are typically used in safety-critical programs, e.g. Boolean logic operations, arithmetic operations) *(2)*. This transformation is discussed in detail in Sections 6–7. The SFC’ to ST’ translation can be implemented using the same principles as the ones used in [12] to represent SFC’ directly using the PLCverif intermediate model *(3)*. Finally, ST’ is a subset of STr’, thus it does not need any further transformation step *(4)*. The STr’ code is the input for the verification model generation.

The PLC *interrupts* were not targeted in this paper. Certain PLCs may use them, interrupting the execution of the main program. A certain IL’ instruction may be atomic, but the corresponding STr’ representation, comprising several statements will not be atomic. This might cause concurrency problems and discrepancies between the two representations of the code. However, if this is critical, a locking mechanism can be added to the translation. Although the IEC 61131 standard does not define any locking mechanism, it is defined for the Siemens ST’ language via the available system function blocks. To fully support the concurrent behaviour, further work will be necessary.

6 Mapping IL' to ST'

We recall that due to our motivation it is not enough to prove the existence of ST' code with an equivalent behaviour for any IL' program. We also need a precise description of the translation to be performed. This section describes the translation from IL' to STr' in detail.

The IL' to STr' translation is crucial in practice, even though IL' is not used for the development of complex programs at CERN. However, the safety-critical programs of fail-safe PLCs can only be written in LD' or FBD' language according to the Siemens guidelines. The programs written in those graphical languages cannot be exported directly, they are converted to IL' first using the Siemens development tool. Therefore the safety-critical programs are accessible only in IL' language. In those cases the usage of error-prone features (e.g. floating-point data types, arrays, structures, pointers; see [27, Sec. 5.1.3] for complete definition) are prohibited, that is why we focus on the instructions in the following categories of [23]: bit logic instructions, simple load and transfer instructions, integer math instructions and comparison instructions (for integers). Contrarily, the instructions in the following groups of [23] are not discussed in this paper: counter instructions, shift and rotate instructions, floating-point math instructions, program control instructions.

As mentioned previously, IL' is a low-level, assembly-like language. It provides access directly to registers, the arithmetic operations are performed on values stored in accumulators, the conditional behaviour is represented by jumps. However, it provides some higher-level features, such as function calls: the user does not have to deal with e.g. putting the parameters of a function call to the stack. Therefore while IL' is commonly considered as an assembly-like language, it resembles more to the bytecodes of managed languages (e.g. Common Intermediate Language of .NET or the Java bytecode).

Another main difference between IL' and the typical low-level instruction sets is the special support for logical operations. PLCs often deal with Boolean logic, therefore dedicated registers (status bits, stacks) and instructions are available to facilitate these operations.

In this section we overview the memory model of the PLCs (Section 6.1). Then we discuss the IL' semantics in Section 6.2 and the representation of the IL' instructions in STr' (Section 6.3). A method for the correctness proof of this translation is given after, in Section 7.

6.1 Memory Model of Siemens PLCs

In this subsection we overview the key features of the memory model that affect the IL' to STr' translation.

General-Purpose Memory. The Siemens PLCs have a globally accessible general-purpose memory. The so-called *bit memory* is a non-structured memory which needs no allocation. *Data blocks* are statically allocated, structured memory blocks. In non-safety settings all these memory parts support direct, indirect or symbolic addressing. To simplify the discussion, we will mainly consider symbolic addressing in the following (i.e. accessing variables via their names instead of their relative or absolute addresses).

Input/Output Memory. The physical inputs and outputs of the PLC are mapped to special ranges of the memory. The speciality of this memory region is that the input values are only sampled at the beginning of the PLC cycles, and the values stored in the output memory will only be assigned to the physical outputs at the end of the PLC cycles. In other aspects these memory ranges behave similarly to the general-purpose memory.

Besides the general-purpose memory, PLC programs use various status bits, accumulators and special-purpose stacks. As discussed before, they are referred simply as *registers*. The following overview of registers is based on [25].

Status Word. Most of the IL' instructions read and write the status word (STW). The status word consists of status bits, as follows [25, p. 13].

- $\overline{\text{FC}}$ (not first computation, bit 0) indicates if there was already any Boolean operation. If it is false, the value of RLO (see below) should not be taken into account. In the following, we will refer to this register as nFC.

- RLO (result of logic operation, bit 1) stores the result of the previous logic operations.

- STA (status, bit 2) stores the Boolean status of the previous operation. It has no effect on the execution of the program, used only for diagnostic purposes [2, Sec. 15.1].

- OR (or, bit 3) stores auxiliary data for the "and before or" operation.

- OV (overflow, bit 5) indicates an overflow occurred in an arithmetic operation.

- Without discussing here their intuitive meaning, the rest of the status bits are the following: OS (stored overflow, bit 4); CC0, CC1 (condition codes, bit 6–7); BR (binary result, bit 8).

Accumulators. To allow binary arithmetic operations, Siemens PLCs use two accumulators, ACCU1 and ACCU2. The size of the accumulators is 32 bits, but their lower and upper 16 bits can be addressed directly (ACCUx-L and ACCUx-H, respectively), just as the individual bytes [25, p. 11]. Certain PLCs have two additional accumulators which might alter the arithmetic operations [2, p. 198].

Nesting Stacks. In order to facilitate the complex Boolean operations, a so-called nesting stack is defined. Each entry of this stack can store a partial result of a logic operation, and the code of the operation to be performed after the stack entry is popped. This makes easy to represent complex Boolean operations with parentheses in IL'. For example, the $r \leftarrow (a \vee b) \wedge (c \vee d) \wedge e$ operation (in ST': `r := (a OR b) AND (c OR d) AND e;`) can be represented as follows:

```
 1  A(        // pushing nesting stack entry
 2  O    a    // OR operation
 3  O    b    // OR operation
 4  )         // pop nesting stack entry and perform AND operation
 5  A(        // pushing nesting stack entry
 6  O    c    // OR operation
 7  O    d    // OR operation
 8  )         // pop nesting stack entry and perform AND operation
 9  A    e    // AND operation
10  =    r    // store result in variable r
```

The `A(` instruction in line 5 stores the current RLO value $(a \vee b)$ and the operation to be performed (`AND`) on the nesting stack. As the nFC is set to false by `A(`, a new logic computation is started. Therefore the value of RLO will be $c \vee d$ after the execution of line 7. When the `)` instruction is executed, the topmost nesting stack entry is popped and the stored `AND` operation will be performed on the stored RLO value (which is $a \vee b$) and the current RLO value (which is $c \vee d$). This will result in a new value for RLO: $(a \vee b) \wedge (c \vee d)$.

Each nesting stack entry contains six bits: three saved status bits and three bits to encode a Boolean operation (so-called function code). We will denote the three status bits of a nesting stack entry by nsRLO, nsOR, nsBR, and the function code bits as nsFC2, nsFC1, nsFC0. To refer to the different entries of the nesting stack, we will use the [ ] indexing operator. For example, nsRLO[1] refers to the RLO bit of the first (topmost) entry of the nesting stack. The nesting stack may contain up to 7 entries [23, Sec. 2.2], indexed by us from 1 (most recent entry) to 7 (least recent entry).

Additional Registers and Stacks. For completeness, we briefly mention some additional registers of Siemens PLCs which will not be discussed in detail in the following.

- AR1 and AR2 are two address registers, providing a base address for indirect addressing.
- The interrupt stack stores the contents of accumulators and address registers when the execution is interrupted by a higher-priority organization block [26, Sec. 27.2.3.4].
- The local stack (L stack) and the block stack (B stack) store the local data of function or function block calls and the return addresses [26, Sec. 27.2.3].
- The MCR (master control relay) bit alters the behaviour of certain instructions which are modifying the values stored in the memory. Its value

can be stored in a special stack and it is possible to have special "MCR zones" [2, p. 232]. As the incorrect usage of MCR zones can lead to errors [2, p. 232], we treat only a simplified case without MCR zones, handling MCR only as a bit register.

6.2 Syntax and Semantics of IL' Instructions

The syntax of the IL' language is rather simple. The body of an IL' program consists of *statements*. Each statement has a *label* (optional), an *instruction* and a *parameter* (depending on the instruction). For some instructions no parameter is needed, for others a symbolic name, a memory address, a label, a function or a constant is required. Without going into formal details, an intuitive syntax definition can be the following:

```
<IL_program> ::= <IL_statement>+;
<IL_statement> ::= [<label> ':'] <IL_instruction> [<parameter>] '\n';
<IL_instruction> ::= 'A' | 'O' | 'X' | 'SET' | 'CLR' | '=' | ...;
```

The instructions and their parameters are defined in [24]. That document typically gives the following information for each instruction:

- The accepted parameter types,
- The informal description of the semantics of the instruction,
- The effect of the instruction on the status word: which bits are written by the instruction (if a certain bit is always set to 0 or 1 by the instruction, this fact is stated, but if the bit value is conditional, the exact formula is not defined),
- An example.

Some further information is available about the IL' instructions in [23, 25]. The latter documentation defines for each instruction on which status bits do they depend.

However, the authors are not aware of any official documentation that describes the precise semantics of each IL' instruction. Obviously, this knowledge is necessary for the formal verification of IL' programs. Therefore we have to propose a method to discover the precise semantics.

Method to Determine the Precise Semantics. In order to discover the precise semantics, we execute the instructions in all possible combinations, i.e. checking the results of each instruction in each possible situation. Of course, testing the behaviour of each instruction with each possible memory content is not feasible. However, each instruction depends only on certain registers and certain parts of the memory. These dependencies are defined in the description or status word influence part of [23, 25]. It is also defined (or can be assumed based on the description), which registers and memory locations might be altered by the execution of a certain instruction[5]. Reproducing all possible combinations of the registers and parameters that affect a specific instruction and checking the new values of the altered affected registers and memory locations seem to be feasible.

On certain PLCs it is possible to set up the status word to a given value using the "T STW" statement. It is also possible to determine the current values of the status bits and registers on a breakpoint using a real PLC or the official Siemens PLC simulator software[6]. Therefore we can generate test programs which execute a given instruction for each interesting valuation. *In summary, we determine the precise semantics by checking for each state (for each variable and register valuations) which new state can be reached by executing a given statement. This is performed using systematically produced test programs which set up each fundamentally different state one-by-one, then execute the IL' instruction under analysis.*

Example. We use the A(instruction as an example to present the method. According to the documentation, "*A((AND nesting open) saves the RLO and OR bits and a function code into the nesting stack. A maximum of seven nesting stack entries are possible.*" [24]. According to [25], the A(instruction depends on the BR, OR, RLO and nFC status bits and it sets the STA bit to true, and the OR and nFC bits to false. The instruction does not depend on any other status bit and it does not modify any other status bit. However, it may depend or affect other data. Based on the informal description, the A(modifies the contents of the nesting stack.

In order to determine the semantics of the A(, all possible combinations of BR, OR, RLO and nFC status bits should be reproduced by a test program, then the resulting values of STA, OR and nFC registers and the nesting stack should be checked. We have generated an IL' test code for the A(instruction. The IL' code snippet corresponding to a check for a single valuation can be seen in Listing 3. This specific code snippet can help us to determine the behaviour of the instruction when the BR bit is false, and the OR, RLO and nFC bits are true.

By generating and performing similar checks for all 2^4 combinations, the data in Table 3 can be obtained. Row 8 of the table was determined by executing the test program in Listing 3. According to the documentation, the STA, OR and nFC bits are set to constant values unconditionally. However, we have already observed mistakes and contradictions in the official documentations, thus it is worth to check the values of STA, OR and nFC too. In the current case the status bit values observed after the execution matched the defined values, therefore

[5] It is precisely defined which status bits can be modified by the instruction, but the same information is not given for other registers or memory locations. However, we assume that an instruction with a variable parameter does not access or modify any other variables if not described explicitly.

[6] It worth to note that technical limitations should be taken into account. For example, according to [23, p. 8-6], in the Siemens S7-300 series PLCs the access to the status word is limited. These limitations does not apply to the S7-400 series PLCs.

```
    L   2#00001011  // BR=0, OR=1, RLO=1, nFC=1
    T   STW
    A(
    NOP 0           // breakpoint here to check the result
    )               // to restore the empty nesting stack
    NOP 0
```

Listing 3: Test code to determine the semantics of the `A(` instruction.

Table 3: Behaviour of the `A(` instruction

Before execution				After execution (new nesting stack entry)			
BR	OR	RLO	nFC	nsBR	nsOR	nsRLO	nsFC2,1,0
0	0	0	0	0	0	1	0,0,0
0	0	0	1	0	0	0	0,0,0
0	0	1	0	0	0	1	0,0,0
0	0	1	1	0	0	1	0,0,0
0	1	0	0	0	0	1	0,0,0
0	1	0	1	0	1	0	0,0,0
0	1	1	0	0	0	1	0,0,0
0	1	1	1	0	1	1	0,0,0
1	0	0	0	1	0	1	0,0,0
1	0	0	1	1	0	0	0,0,0
1	0	1	0	1	0	1	0,0,0
1	0	1	1	1	0	1	0,0,0
1	1	0	0	1	0	1	0,0,0
1	1	0	1	1	1	0	0,0,0
1	1	1	0	1	0	1	0,0,0
1	1	1	1	1	1	1	0,0,0

they are omitted from the table. Based on Table 3 and the documentation the effects of the `A(` instruction can be summarised:

- It sets the OR and nFC status bits to 0,
- It sets the STA status bit to 1,
- It creates a new nesting stack entry, where:
  - The value of OR bit is $\mathsf{OR} \land \mathsf{nFC}$,
  - The value of RLO bit is $\mathsf{RLO} \lor \neg\mathsf{nFC}$,
  - The value of BR bit is BR[7],
  - The value of function encoding is $(0, 0, 0)$, corresponding to the "`A(`" instruction [23], and
- It pushes this new nesting stack entry into the nesting stack.

It is worth to be noted that contrarily to the straightforward meaning of the description, based on our systematic checks the `A(` instruction does not store the exact values of the RLO and the OR bits (i.e. the nsRLO does not equal to RLO in every case). This demonstrates that the IL' to STr' translation cannot rely only on the informal description of the instructions.

6.3 STr' Translation of IL' Instructions

As discussed previously, there is another challenge in the IL' to ST' transformation besides the lack of precise IL' semantics: the IL' language has lower level instructions than the ST', the IL' instructions typically access and modify the registers directly. In our translation method, following the principles discussed previously, we represent IL' programs in STr' by emulating the registers with local variables, and by making the effects of the IL' instructions explicit. In order to avoid the name collisions, the variables representing registers will be prefixed by "`__`". The ST' definition normally forbids the usage of double underscores as prefix.

Following these principles and conventions, using the knowledge gathered by performing the systematic analysis of the IL' instruction semantics, it is possible to give an STr' representation for each IL' instruction. Table 4 shows the STr' representation of the basic bit logic IL' instructions. After, Table 5 shows the representation of the basic arithmetic instructions. Note that in Table 5 instructions on 32-bit integers are assumed, but the instructions for smaller data types can be defined similarly. The more advanced logic instructions, relying on the nesting stack, are in Table 6.

[7] We were not able to observe directly the BR value of the nesting stack entry. Instead, we have checked the value of the BR status bit after the `)` instruction. The result (i.e. the BR bit of the nesting stack entry equals to the value of the BR status bit) is in accordance with our expectations based on the informal descriptions.

Table 4: STr' representation of simple bit logic IL' instructions

IL'	STr' equivalent

A *var1*	IF __NFC THEN __RLO:=__RLO AND (*var1* OR __OR); ELSE __RLO:=*var1* OR __OR; END_IF;
	__STA:=*var1*; __NFC:=TRUE;

AN *var1*	IF __NFC THEN __RLO:=__RLO AND (NOT *var1* OR __OR); ELSE __RLO:=NOT *var1* OR __OR; END_IF;
	__STA:=*var1*; __NFC:=TRUE;

O *var1*	IF __NFC THEN __RLO:=__RLO OR *var1*; ELSE __RLO:=*var1*; END_IF;
	__OR:=FALSE; __STA:=*var1*; __NFC:=TRUE;

ON *var1*	IF __NFC THEN __RLO:=__RLO OR (NOT *var1*); ELSE __RLO:=NOT *var1*; END_IF;
	__OR:=FALSE; __STA:=*var1*; __NFC:=TRUE;

X *var1*	IF __NFC THEN __RLO:=__RLO XOR *var1*; ELSE __RLO:=*var1*; END_IF;
	__OR:=FALSE; __STA:=*var1*; __NFC:=TRUE;

XN *var1*	IF __NFC THEN __RLO:=__RLO XOR (NOT *var1*); ELSE __RLO:=NOT *var1*; END_IF;
	__OR:=FALSE; __STA:=*var1*; __NFC:=TRUE;

O	__STA:=TRUE; __OR:=__NFC AND (__OR OR __RLO); __NFC:=__RLO AND __NFC;

= *var1*	IF __MCR THEN *var1*:=__RLO; ELSE *var1*:=FALSE; END_IF;
	__OR:=FALSE; __STA:=*var1*; __NFC:=FALSE;

S *var1*	IF __MCR AND __RLO THEN *var1*:=TRUE; END_IF; __OR:=FALSE; __STA:=*var1*; __NFC:=FALSE;
R *var1*	IF __MCR AND __RLO THEN *var1*:=FALSE; END_IF; __OR:=FALSE; __STA:=*var1*; __NFC:=FALSE;

FP *var1*	__OR:=FALSE; __STA:=__RLO; __NFC:=TRUE;
	IF NOT *var1* AND __RLO THEN
	    var1:=__RLO; __RLO:=TRUE; *//rising edge detected*
	ELSE
	    var1:=__RLO; __RLO:=FALSE;
	END_IF;

FN *var1*	__OR:=FALSE; __STA:=__RLO; __NFC:=TRUE;
	IF *var1* AND NOT __RLO THEN
	    var1:=__RLO; __RLO:=TRUE; *//falling edge detected*
	ELSE
	    var1:=__RLO; __RLO:=FALSE;
	END_IF;

NOT	__RLO:=NOT __RLO OR __OR; __STA:=TRUE; (*It does not set NFC.*)
CLR	__RLO:=FALSE; __OR:=FALSE; __STA:=FALSE; __NFC:=FALSE;
SET	__RLO:=TRUE; __OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;
SAVE	__BR:=__RLO;
MCRA	__MCR:=TRUE;
MCRD	__MCR:=FALSE;

Table 5: STr' representation of load and transfer, and integer math IL' instructions

IL'	STr' equivalent
L *var1*	__ACCU2 := __ACCU1; __ACCU1 := *var1*;
T *var1*	IF __MCR THEN *var1*:=__ACCU1; ELSE *var1*:=0; END_IF;

>D	__OR:=FALSE; __NFC:=TRUE; __RLO:=(__ACCU1<__ACCU2);
	__CC0:=(__ACCU1>__ACCU2); __CC1:=(__ACCU1<__ACCU2);
	__OV:=FALSE; __OR:=FALSE; __NFC:=TRUE; __STA:=__RLO;

>=D	__OR:=FALSE; __NFC:=TRUE; __RLO:=(__ACCU1<=__ACCU2);
	__CC0:=(__ACCU1>__ACCU2); __CC1:=(__ACCU1<__ACCU2);
	__OV:=FALSE; __OR:=FALSE; __NFC:=TRUE; __STA:=__RLO;

<D	__OR:=FALSE; __NFC:=TRUE; __RLO:=(__ACCU1>__ACCU2);
	__CC0:=(__ACCU1>__ACCU2); __CC1:=(__ACCU1<__ACCU2);
	__OV:=FALSE; __OR:=FALSE; __NFC:=TRUE; __STA:=__RLO;

<=D	__OR:=FALSE; __NFC:=TRUE; __RLO:=(__ACCU1>=__ACCU2);
	__CC0:=(__ACCU1>__ACCU2); __CC1:=(__ACCU1<__ACCU2);
	__OV:=FALSE; __OR:=FALSE; __NFC:=TRUE; __STA:=__RLO;

==D	__OR:=FALSE; __NFC:=TRUE; __RLO:=(__ACCU1=__ACCU2);
	__CC0:=(__ACCU1>__ACCU2); __CC1:=(__ACCU1<__ACCU2);
	__OV:=FALSE; __OR:=FALSE; __NFC:=TRUE; __STA:=__RLO;

Table 6: STr' representation of nesting stack operations

IL'	STr' equivalent

A(	__nsRLO[7]:=__nsRLO[6]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO OR NOT __NFC;
	__nsOR[7] := __nsOR[6]; ... __nsOR[2] := __nsOR[1]; __nsOR[1]:=__OR AND __NFC;
	__nsBR[7] := __nsBR[6]; ... __nsBR[2] := __nsBR[1]; __nsBR[1]:=__BR;
	__nsFC2[7]:=__nsFC2[6]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=FALSE;
	__nsFC1[7]:=__nsFC1[6]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=FALSE;
	__nsFC0[7]:=__nsFC0[6]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=FALSE;
	__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;

AN(	__nsRLO[7]:=__nsRLO[6]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO OR NOT __NFC;
	__nsOR[7] := __nsOR[6]; ... __nsOR[2] := __nsOR[1]; __nsOR[1]:=__OR AND __NFC;
	__nsBR[7] := __nsBR[6]; ... __nsBR[2] := __nsBR[1]; __nsBR[1]:=__BR;
	__nsFC2[7]:=__nsFC2[6]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=FALSE;
	__nsFC1[7]:=__nsFC1[6]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=FALSE;
	__nsFC0[7]:=__nsFC0[6]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=TRUE;
	__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;

O(	__nsRLO[7]:=__nsRLO[6]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO AND __NFC;
	__nsOR[7] := __nsOR[6]; ... __nsOR[2] := __nsOR[1]; __nsOR[1]:=FALSE;
	__nsBR[7] := __nsBR[6]; ... __nsBR[2] := __nsBR[1]; __nsBR[1]:=__BR;
	__nsFC2[7]:=__nsFC2[6]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=FALSE;
	__nsFC1[7]:=__nsFC1[6]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=TRUE;
	__nsFC0[7]:=__nsFC0[6]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=FALSE;
	__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;

ON(	__nsRLO[7]:=__nsRLO[6]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO AND __NFC;
	__nsOR[7] := __nsOR[6]; ... __nsOR[2] := __nsOR[1]; __nsOR[1]:=FALSE;
	__nsBR[7] := __nsBR[6]; ... __nsBR[2] := __nsBR[1]; __nsBR[1]:=__BR;
	__nsFC2[7]:=__nsFC2[6]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=FALSE;
	__nsFC1[7]:=__nsFC1[6]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=TRUE;
	__nsFC0[7]:=__nsFC0[6]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=TRUE;
	__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;

X(	__nsRLO[7]:=__nsRLO[6]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO AND __NFC;
	__nsOR[7] := __nsOR[6]; ... __nsOR[2] := __nsOR[1]; __nsOR[1]:=FALSE;
	__nsBR[7] := __nsBR[6]; ... __nsBR[2] := __nsBR[1]; __nsBR[1]:=__BR;
	__nsFC2[7]:=__nsFC2[6]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=TRUE;
	__nsFC1[7]:=__nsFC1[6]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=FALSE;
	__nsFC0[7]:=__nsFC0[6]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=FALSE;
	__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;

XN(	__nsRLO[7]:=__nsRLO[6]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO AND __NFC;
	__nsOR[7] := __nsOR[6]; ... __nsOR[2] := __nsOR[1]; __nsOR[1]:=FALSE;
	__nsBR[7] := __nsBR[6]; ... __nsBR[2] := __nsBR[1]; __nsBR[1]:=__BR;
	__nsFC2[7]:=__nsFC2[6]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=TRUE;
	__nsFC1[7]:=__nsFC1[6]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=FALSE;
	__nsFC0[7]:=__nsFC0[6]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=TRUE;
	__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;

)	__OR:=__nsOR[1]; __NFC:=TRUE; __STA:=TRUE; __BR:=__nsBR[1];
	IF (NOT __nsFC2[1] AND NOT __nsFC1[1] AND NOT __nsFC0[1])THEN
	    __RLO:=(__nsRLO[1] AND __RLO)OR __OR[1]; *//A(instruction, FC=(0,0,0)*
	ELSIF (NOT __nsFC2[1] AND NOT __nsFC1[1] AND __nsFC0[1])THEN
	    __RLO:=(__nsRLO[1] AND NOT __RLO)OR __OR[1]; *//AN(instruction, FC=(0,0,1)*
	ELSIF (NOT __nsFC2[1] AND __nsFC1[1] AND NOT __nsFC0[1])THEN
	    __RLO:=__nsRLO[1] OR __RLO; *//O(instruction, FC=(0,1,0)*
	ELSIF (NOT __nsFC2[1] AND __nsFC1[1] AND __nsFC0[1])THEN
	    __RLO:=__nsRLO[1] OR (NOT __RLO); *//ON(instruction, FC=(0,1,1)*
	ELSIF (__nsFC2[1] AND NOT __nsFC1[1] AND NOT __nsFC0[1])THEN
	    __RLO:=__nsRLO[1] XOR __RLO; *//X(instruction, FC=(1,0,0)*
	ELSE
	    __RLO:=__nsRLO[1] XOR (NOT __RLO); *//XN(instruction, FC=(1,0,1)*
	END_IF;
	// Stack pop
	__nsRLO[1]:=__nsRLO[2]; ... __nsRLO[6]:=__nsRLO[7]; __nsRLO[7]:=FALSE;
	__nsOR[1] := __nsOR[2]; ... __nsOR[6] := __nsOR[7]; __nsOR[7]:=FALSE;
	__nsBR[1] := __nsBR[2]; ... __nsBR[6] := __nsBR[7]; __nsBR[7]:=FALSE;
	__nsFC2[1]:=__nsFC2[2]; ... __nsFC2[6]:=__nsFC2[7]; __nsFC2[7]:=FALSE;
	__nsFC1[1]:=__nsFC1[2]; ... __nsFC1[6]:=__nsFC1[7]; __nsFC1[7]:=FALSE;
	__nsFC0[1]:=__nsFC0[2]; ... __nsFC0[6]:=__nsFC0[7]; __nsFC0[7]:=FALSE;

7 Proving the Correctness of the IL' to ST' Translation

In this section we present a method to prove the correctness of the translation from IL' to STr', i.e. if the STr' equivalents have the same behaviour as the corresponding IL' instructions according to the experiments. For this, we perform the following steps:

- Defining the formal semantics of STr' (Section 7.1),
- Defining the formal semantics of IL' (Section 7.2), and
- Giving a proof strategy to show the equivalence (Section 7.3).

The following discussion focuses on the principles of this proof strategy and does not provide a complete correctness proof.

7.1 Formal Semantics for STr'

In this section we draw up an operational semantics for the ST' (STr') language. We will denote the context of an ST' statement by σ. This is a function $\sigma\colon V \to D$, i.e. a function that assigns a value from pre-defined domains to each defined variable. The program P executed from an initial context σ_0 results in σ_1 which will determine the values of the physical outputs and the initial values of retained variables for the following PLC cycle.

At the beginning of the program execution, each variable has an explicitly or implicitly defined default value (the implicit default values are 0 or false). Then in the beginning of each cycle the non-local variables keep their previously set values, while the initial values of local variables are undefined. The execution ends when the final configuration $(\langle \textbf{skip;} \rangle, \sigma)$ is reached ("**skip;**" denotes that there is no more program code to be executed).

An intuitive formalisation of the ST' statements' semantics is presented in Figure 3. This is a small-step semantics, i.e. it defines the operation of a program step by step. The semantics definition in Figure 3 consists of a set of inference rules. Each inference rule consists of some (zero or more) premises (above the horizontal line) and a conclusion (below the line). The operation of a given program with a given initial context can be determined by applying the inference rules one after another until the final configuration $(\langle \textbf{skip;} \rangle, \sigma)$ is reached.

Note that the expression evaluation is not presented in Figure 3 in detail (only the `OR` and `AND` operators are defined as illustration), furthermore only the variable assignment and the `IF` statements are presented. In the rules v_1 denotes a variable, e_1, e_2 are expressions, c_1, c_2 are constant values, s_1 and s_2 are ST' statements or ST' statement lists. We distinguish between arithmetic or logic evaluation (denoted by $\longrightarrow_a$) and single-step program evaluations (denoted by $\longrightarrow$). If a program evaluation is possible in more steps we will denote it by $\longrightarrow^*$. Let us denote by $\sigma[v_1 \mapsto c_1]$ the function that is equivalent to σ except that $\sigma(v_1) = c_1$. Formally:

$$\sigma[v_1 \mapsto c_1](x) = \begin{cases} \sigma(x) & \text{if } x \neq v_1 \\ c_1 & \text{if } x = v_1. \end{cases}$$

For the sake of readability, we will use the following, comma-separated format too: $\sigma[v_1 \mapsto c_1, \ldots, v_n \mapsto c_n] = \big((\sigma[v_1 \mapsto c_1]) \cdots\big)[v_n \mapsto c_n]$.

$$\frac{\sigma(v_1) = c_1}{(v_1, \sigma) \longrightarrow_a c_1}\ \text{ST' \textsc{Variable value}}$$

$$\frac{(e_1, \sigma) \longrightarrow_a c_1 \qquad (e_2, \sigma) \longrightarrow_a c_2}{(\langle e_1 \texttt{ OR } e_2\rangle, \sigma) \longrightarrow_a c_1 \vee c_2}\ \text{ST' OR \textsc{expression}}$$

$$\frac{(e_1, \sigma) \longrightarrow_a c_1 \qquad (e_2, \sigma) \longrightarrow_a c_2}{(\langle e_1 \texttt{ AND } e_2\rangle, \sigma) \longrightarrow_a c_1 \wedge c_2}\ \text{ST' AND \textsc{expression}}$$

$$\frac{(\langle s_1\,\texttt{;}\rangle, \sigma) \longrightarrow (\langle \textbf{skip}\texttt{;}\rangle, \sigma')}{(\langle s_1\,\texttt{;}\,s_2\,\texttt{;}\rangle, \sigma) \longrightarrow (\langle s_2\,\texttt{;}\rangle, \sigma')}\ \text{ST' \textsc{Sequence}}$$

$$\frac{(e_1, \sigma) \longrightarrow_a c_1}{(\langle v_1\texttt{:=}e_1\texttt{;}\rangle, \sigma) \longrightarrow (\langle \textbf{skip}\texttt{;}\rangle, \sigma[v_1 \mapsto c_1])}\ \text{ST' \textsc{Assignment}}$$

$$\frac{(e_1, \sigma) \longrightarrow_a \top}{(\langle \texttt{ IF } e_1 \texttt{ THEN } s_1 \texttt{ ELSE } s_2 \texttt{ END\_IF; } \rangle, \sigma) \longrightarrow (s_1, \sigma)}\ \text{ST' \textsc{If} (1)}$$

$$\frac{(e_1, \sigma) \longrightarrow_a \bot}{(\langle \texttt{ IF } e_1 \texttt{ THEN } s_1 \texttt{ ELSE } s_2 \texttt{ END\_IF; } \rangle, \sigma) \longrightarrow (s_2, \sigma)}\ \text{ST' \textsc{If} (2)}$$

Figure 3: Simplified ST' semantics

7.2 Formal Semantics for IL'

In this section the goal is to describe the formal semantics of IL', similarly to the semantics of ST' in the previous section. We will denote the context of an ST' statement by σ, ρ. The first function is $\sigma\colon V \to D$ that assigns a value from pre-defined domains to each defined variable, similarly to the ST' semantics. The second function is $\rho\colon R \to D$ that assigns values to the set of registers $R = \{\mathsf{MCR}, \mathsf{nFC}, \mathsf{RLO}, \mathsf{STA}, \ldots, \mathsf{nsRLO[1]}, \mathsf{nsOR[1]}, \ldots \mathsf{nsFC0[7]}\}$.

At the beginning of the program execution, each variable has an explicitly or implicitly defined default value in σ (the implicit default values are 0 or false). Then at the beginning of each cycle the non-local variables keep their previously set values, while the initial values of local variables are undefined. The registers are initialised to their default values at the beginning of each cycle. The default values of the registers are false, except for MCR, RLO and STA which are initialised to true at the beginning of each cycle.

We define the basics of semantics, such as the variable and register values, or the semantics of the sequence of instructions, as follows in Figure 4. In the rules v_1 is a variable, c_1 is a constant value, s_1 and s_2 are IL' statements or IL' statement lists.

$$\frac{\sigma(v_1) = c_1}{(v_1, \sigma, \rho) \longrightarrow_a c_1} \text{ IL' \textsc{Variable value}}$$

$$\frac{\rho(r_1) = c_1}{(r_1, \sigma, \rho) \longrightarrow_a c_1} \text{ IL' \textsc{Register value}}$$

$$\frac{(\langle s_1 \rangle, \sigma, \rho) \longrightarrow (\langle \mathbf{skip} \rangle, \sigma', \rho')}{(\langle s_1 \quad s_2 \rangle, \sigma, \rho) \longrightarrow (\langle s_2 \rangle, \sigma', \rho')} \text{ IL' \textsc{Sequence}}$$

Figure 4: Simplified IL' semantics

Formalising the Discovered IL' Semantics. As it was discussed previously, the semantics of the IL' instructions are not defined precisely. In our method we conduct systematic experiments to determine the semantics of the IL' instructions in each possible configuration. The IL' semantics is known only through these observed semantics. This was summarised in tables, similarly to Table 3. These tables can systematically be transformed into inference rules. Each row of the table can be represented as a single inference rule, therefore the semantics of a given IL' instruction will be formalised as a set of inference rules, one for each possible initial configuration. This is demonstrated by the following example.

Example. Here we will use a simple "or" operation `O v1` as an example, where `v1` is a variable. The observed semantics of `O v1`, obtained through a series of tests as described before can be seen in Table 7.

Table 7: Observed IL' semantics for `O v1`

Before execution			After execution				
RLO	nFC	v1	v1	OR	STA	RLO	nFC
0	0	0	0	0	0	0	1
0	0	1	1	0	1	1	1
0	1	0	0	0	0	0	1
0	1	1	1	0	1	1	1
1	0	0	0	0	0	0	1
1	0	1	1	0	1	1	1
1	1	0	0	0	0	1	1
1	1	1	0	0	1	1	1

Each row of this table describes the semantics of the `O v1` statement with different preconditions. For example, row 2 defines the following formal semantics:

$$\frac{\rho(\mathsf{RLO}) = \bot \quad \rho(\mathsf{nFC}) = \bot \quad \sigma(v_1) = \top}{(\langle \mathtt{O}\ v_1 \rangle, \sigma, \rho) \longrightarrow (\langle \mathbf{skip} \rangle, \sigma, \rho[\mathsf{OR} \mapsto \bot, \mathsf{STA} \mapsto \top, \mathsf{RLO} \mapsto \top, \mathsf{nFC} \mapsto \top])} \text{ IL' O } v_1 \text{ (2)}$$

After formalising the ST' and IL' semantics, the only remaining step is to prove that the suggested STr' representations of the IL' statements will provide the same results. This proof will be drawn up in the next section.

```
IF __NFC THEN
    __RLO:=__RLO OR v1;
ELSE
    __RLO:=v1;
END_IF;
__OR:=FALSE;
__STA:=v1;
__nFC:=TRUE;
```

Listing 4: STr' equivalent of the `O v1` IL' statement

7.3 Strategy for the Correctness Proof

Now it is possible to define formally the correctness of the IL' to ST' translation. Formally, the goal is to prove the following:

$$((P_{IL},\sigma_1,\rho_1) \longrightarrow^* (\langle\textbf{skip}\rangle,\sigma_2,\rho_2)) \Longrightarrow ((P_{ST},\sigma'_1) \longrightarrow^* (\langle\textbf{skip;}\rangle,\sigma'_2)),$$

where P_{ST} is the STr' representation of the IL' code P_{IL}, and σ'_i is the representation of σ_i, ρ_i such that:

$$\sigma'_i(x) = \begin{cases} \sigma_i(x) & \text{if } x \text{ is a real variable} \\ \rho_i(y) & \text{if } x \text{ is the STr' variable representing the register } y \text{ } (\mathsf{\_\_y}{=}x). \end{cases}$$

The program P_{IL} is a sequence of IL statements. Based on the ST' Sequence and IL' Sequence inference rules of the semantics definitions discussed before, it is enough to prove that the behaviour of each IL' instruction corresponds to their STr' representations' behaviour. We have to show that the proposed STr' equivalent of a certain IL' instruction provides the same semantics as the original IL' instruction. As the semantics of a given IL' instruction is formalised as a set of inference rules (as discussed in the previous section), the goal is to show that for each IL' inference rule given the same premises (equivalent initial contexts), given the STr' representation, and using the inference rules of the ST' semantics, the reached final configuration of the STr' program corresponds to the final configuration of the IL' instruction. This is demonstrated by the following example.

Example. For example, based on Table 7, the STr' equivalent presented in Listing 4 can be proposed for `O v1`.

For each row of Table 7 it is possible to formally prove based on the defined STr' semantics that the proposed STr' equivalent provides the same result. The inference tree in Figure 5 proves that the above STr' code matches the previously discussed IL' O v_1 (2) semantic rule. This proof shows that from the same premises ($\sigma(\mathsf{\_\_nFC}) = \bot, \sigma(\mathsf{\_\_RLO}) = \bot, \sigma(v_1) = \top$), by applying the inference rules of the ST' semantics, the reached configuration corresponds to the one reached by the execution of the IL' statement: the context $\sigma'[\mathsf{\_\_OR} \mapsto \bot, \mathsf{\_\_STA} \mapsto \top, \mathsf{\_\_RLO} \mapsto \top, \mathsf{\_\_nFC} \mapsto \top]$ reached by the STr' code emulates the final configuration in the IL' semantics definition ($\rho[\mathsf{OR} \mapsto \bot, \mathsf{STA} \mapsto \top, \mathsf{RLO} \mapsto \top, \mathsf{nFC} \mapsto \top], \sigma$).

By looking at Figure 5 it can also be seen that the result does not depend on the value of the RLO register, therefore the same proof can be used for the IL' O v_1 (6) semantic rule, describing row 6 of Table 7:

$$\frac{\rho(\mathsf{RLO}) = \top \quad \rho(\mathsf{nFC}) = \bot \quad \sigma(v_1) = \top}{(\langle \mathsf{O}\ v_1 \rangle, \sigma, \rho) \longrightarrow (\langle \mathbf{skip} \rangle, \sigma, \rho[\mathsf{OR} \mapsto \bot, \mathsf{STA} \mapsto \top, \mathsf{RLO} \mapsto \top, \mathsf{nFC} \mapsto \top])} \text{ IL' O } v_1 \text{ (6)}$$

8 Application and Outlook to Verification

The motivation of this work was to extend PLCverif with support for more PLC languages. The LD' and FBD' languages are especially interesting, as these are the only languages that can be used to program fail-safe (safety) Siemens PLCs. The only way to access the source code written for Siemens safety PLCs is to export the LD' or FBD' code as IL' code. Therefore the verification tool to be used has to handle programs written in IL'. By establishing an IL' to STr' translation, the PLCverif tool made available for the verification of safety PLC programs.

Complete Verification Workflow. The complete verification workflow of a safety PLC program is depicted in Figure 6. The verification is based on the implementation of the safety logic (written either in LD' or FBD' language) and the informal specification. The LD' or FBD' implementation can be transformed into IL' code by the Siemens development tool. Then, using the IL' to ST' translation principles discussed in this paper, an STr' representation can be generated that has an equivalent behaviour to the original IL' code.

The requirements from the informal specification can be extracted using requirement patterns. Each pattern has a precise textual description with some placeholders, which have to be filled by the user based on the informal specification. Each pattern has a formal representation too based on temporal logic which can be then used by the model checker tool [12].

The original PLCverif workflow [13] is based on an ST' (STr') implementation and some filled requirement patterns. Based on these artefacts an intermediate model (IM) is generated, which is then reduced and translated to the concrete syntax of the chosen model checker. The external model checker tool is executed, after its result is parsed and then presented to the user in a verification report.

Using PLCverif for Safety-Critical Systems. In [8] we have presented a first case study of using PLCverif for safety-critical PLC programs. The target of this work was the verification of a safety logic that is used in the SM18 Cryogenic Test Facility at CERN. This test facility allows to check various properties of superconducting magnets at low temperatures, in vacuum, and at high currents. These magnet tests might be dangerous if certain subsystems are not ready, e.g. high currents should only be applied when the cryogenic system

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\sigma(\mathsf{\_\_nFC}) = \bot}{(\texttt{\_\_NFC}, \sigma) \longrightarrow_a \bot}
}{
\begin{array}{c}
(\langle \texttt{IF \_\_NFC THEN \_\_RLO:=\_\_RLO OR v1;} \\
\texttt{ELSE \_\_RLO:=v1; END\_IF;} \rangle, \sigma) \longrightarrow \\
(\langle \texttt{\_\_RLO:=v1;} \rangle, \sigma)
\end{array}
}
\quad
\dfrac{\sigma(v_1) = \top}{(\texttt{v1}, \sigma) \longrightarrow_a \top}
}{
\begin{array}{c}
(\langle \texttt{IF \_\_NFC THEN \_\_RLO:=\_\_RLO OR v1;} \\
\texttt{ELSE \_\_RLO:=v1; END\_IF;} \rangle, \sigma) \longrightarrow \\
(\langle \mathbf{skip;} \rangle, \sigma[\mathsf{\_\_RLO} \mapsto \top])
\end{array}
}
\quad
\dfrac{.}{
\begin{array}{c}
(\langle \texttt{\_\_OR:=FALSE;} \rangle, \sigma[\mathsf{\_\_RLO} \mapsto \top]) \longrightarrow \\
(\langle \mathbf{skip;} \rangle, \sigma[\mathsf{\_\_OR} \mapsto \bot, \mathsf{\_\_RLO} \mapsto \top])
\end{array}
}
}{
\begin{array}{c}
(\langle \texttt{IF \_\_NFC THEN \_\_RLO:=\_\_RLO OR v1; ELSE \_\_RLO:=v1; END\_IF; \_\_OR:=FALSE;} \rangle, \sigma) \longrightarrow \\
(\langle \mathbf{skip;} \rangle, \sigma[\mathsf{\_\_OR} \mapsto \bot, \mathsf{\_\_RLO} \mapsto \top])
\end{array}
}
\quad
\dfrac{
\dfrac{\sigma(v_1) = \top}{(\texttt{v1}, \sigma) \longrightarrow_a \top}
}{
\begin{array}{c}
(\langle \texttt{\_\_STA:=v1;} \rangle, \sigma[\mathsf{\_\_OR} \mapsto \bot, \mathsf{\_\_RLO} \mapsto \top]) \longrightarrow \\
(\langle \mathbf{skip;} \rangle, \sigma[\mathsf{\_\_OR} \mapsto \bot, \mathsf{\_\_STA} \mapsto \top, \mathsf{\_\_RLO} \mapsto \top])
\end{array}
}
}{
\begin{array}{c}
(\langle \texttt{IF \_\_NFC THEN \_\_RLO:=\_\_RLO OR v1; ELSE \_\_RLO:=v1; END\_IF; \_\_OR:=FALSE; \_\_STA:=v1;} \rangle, \sigma) \longrightarrow \\
(\langle \mathbf{skip;} \rangle, \sigma[\mathsf{\_\_OR} \mapsto \bot, \mathsf{\_\_STA} \mapsto \top, \mathsf{\_\_RLO} \mapsto \top])
\end{array}
}
\quad
\dfrac{.}{
\begin{array}{c}
(\langle \texttt{\_\_NFC:=TRUE;} \rangle, \sigma[\mathsf{\_\_OR} \mapsto \bot, \mathsf{\_\_STA} \mapsto \top, \mathsf{\_\_RLO} \mapsto \top]) \longrightarrow \\
(\langle \mathbf{skip;} \rangle, \sigma[\mathsf{\_\_OR} \mapsto \bot, \mathsf{\_\_STA} \mapsto \top, \mathsf{\_\_RLO} \mapsto \top, \mathsf{\_\_nFC} \mapsto \top])
\end{array}
}
}{
\begin{array}{c}
(\langle \texttt{IF \_\_NFC THEN \_\_RLO:=\_\_RLO OR v1; ELSE \_\_RLO:=v1; END\_IF; \_\_OR:=FALSE; \_\_STA:=v1; \_\_nFC:=TRUE;} \rangle, \sigma) \longrightarrow \\
(\langle \mathbf{skip;} \rangle, \sigma[\mathsf{\_\_OR} \mapsto \bot, \mathsf{\_\_STA} \mapsto \top, \mathsf{\_\_RLO} \mapsto \top, \mathsf{\_\_nFC} \mapsto \top])
\end{array}
}
$$

Figure 5: Proof of semantic equivalence between the STr' representation of `O v1` and the semantic rule IL' O v_1 (2)

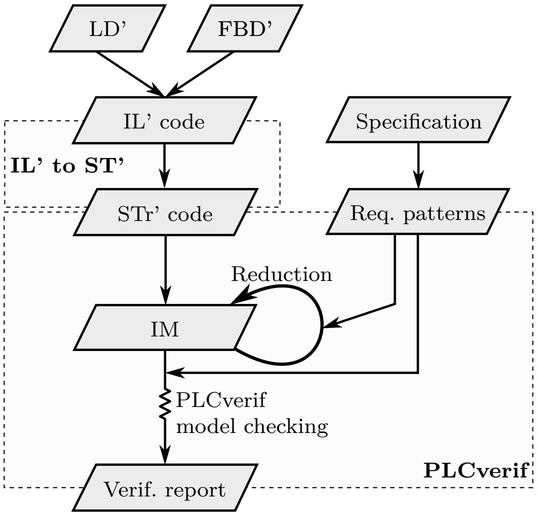


Figure 6: Workflow of checking a safety PLC program (based on [8])

is properly working. The purpose of the safety logic under verification is to ensure the safety of the magnet tests by allowing or forbidding them based on preconditions of the magnet test scenarios.

This safety logic is a significantly large, real safety-critical program. The IL' representation of the implementation consists of 9,500 lines of code, while the generated STr' representation is composed of more than 120,000 statements [8].

This verification case study has demonstrated that model checking can reveal complex issues with moderate effort, before putting the system in production. In this case 12 different problems were identified using the above-described model checking workflow implemented in PLCverif. More details about this case study can be found in [8].

9 Related Work

This is not the first paper aiming to translate programs between PLC languages. Sülflow and Drechsler [29] translated Siemens IL' (STL) programs to SystemC for verification purposes. They have discussed the semantics of Siemens IL', but only to a limited extent, e.g. the nesting logic statements are not targeted. Pavlović et al. targets the formal verification of Siemens IL' programs in [19]. For this reason, they discuss the formal and informal semantics of the Siemens IL' instructions. Both in [29] and [19] one can find translations similar to the ones in Table 4 and 5 of the current paper. However, they did not discuss how could the precise semantics be determined, therefore due to the lack of available semantics definition, they could not handle the nesting stack for example. Besides, they did not target the proof of correctness.

Meulen [17] provides formal semantics for the Siemens IL' (STL) language to use propositional logic for verification. In his thesis the discussion of instructions is more complete than in [19, 29], however the instructions with more complex semantics are not targeted here either.

Sadolewski discussed the translation of IEC 61131 ST programs to C and Why for verification [21, 20]. Similarly, Kabra et al. [16] targeted the ST to C translation. Biha and Blech discusses the formal semantics of IL and LD languages [5, 4]. However, as they translated programs written in IEC 61131 standard languages, their results cannot be directly applied to Siemens PLC programs due to the semantic and syntactic differences. Therefore to the authors' best knowledge no previous work published precise STL semantics that is essential for the formal verification of these programs.

Certain works target specifically the verification of Siemens variants of IL' (STL) programs [3, 19, 18]. However, none of these papers focus specifically on the details of the semantics, e.g. the precise representation of the nesting stack operations.

Awlsim[8] is an open-source simulator for Siemens IL' (STL) programs. The authors claim that it provides a nearly complete, Siemens-equivalent simulation of IL' programs. However, by analysing the source code it seems that the semantics was not determined precisely, and for example the nesting stack handling is simplified compared to what is observed in reality[9].

10 Summary

This paper presented the relations between the different PLC programming languages, both for the standard versions of IEC 61131 and the Siemens implementations. For our practical goals, i.e. to extend PLCverif to support all five Siemens variants of the PLC languages, a good pivot language candidate was proposed: STr' that is the Siemens ST' language emulating register access with variable access for verification purposes. Using STr', PLCverif can efficiently support the verification of low-level languages (IL', FBD', LD'), without modifying the core workflow or decreasing the verification performance of the programs written in ST' language. We have drawn up a translation from IL' language to STr', which is a crucial step towards the verification of safety PLC programs. This translation description included a systematic observation of semantics, the formalisation of semantics and a correctness proof method of the transformation.

[8] `http://bues.ch/cms/automation/awlsim.html`

[9] See for example `https://github.com/mbuesch/awlsim/blob/master/awlsim/core/instructions/insn_ub.py` that implements the behaviour of the `A(` instruction. It can be seen that the complete status word is saved to the nesting stack as it is, however in Section 6.2 it was shown that the `OR` and `RLO` bits are modified before saving them.

References

[1] Nanette Bauer, Ralf Huuck, Ben Lukoschus, and Sebastian Engell. A unifying semantics for sequential function charts. In Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 400–418. Springer, 2004.

[2] Hans Berger. *Automating with STEP 7 in STL and SCL*. Publicis Corporate Publishing, 3rd edition, 2005.

[3] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. Arcade.PLC: A verification platform for programmable logic controllers. In *Proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 338–341. IEEE, 2012.

[4] Sidi Ould Biha. A formal semantics of PLC programs in Coq. In *IEEE 35th Annual Computer Software and Applications Conference*, pages 118–127, 2011.

[5] Jan Olaf Blech and Sidi Ould Biha. On formal reasoning on the semantics of PLC using Coq. Preprint, available online `arXiv:1301.3047 [cs.SE]`, 2013.

[6] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[7] Dániel Darvas, Borja Fernández Adiego, and Enrique Blanco Viñuela. PLCverif: A tool to verify PLC programs based on model checking techniques. In Lou Corvetti, Kathleen Riches, and Volker R.W. Schaa, editors, *Proc. of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*, pages 911–914. JACoW, 2015.

[8] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Formal verification of safety PLC based control software. In Erika Ábrahám and Marieke Huisman, editors, *Integrated Formal Methods*, volume 9681 of *Lecture Notes in Computer Science*, pages 508–522. Springer, 2016.

[9] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Generic representation of PLC programming languages for formal verification. In *Proc. of the 23rd PhD Mini-Symposium*, pages 6–9. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2016.

[10] Mário de Sousa. Proposed corrections to the IEC 61131-3 standard. *Computer Standards & Interfaces*, 32(5-6):312–320, 2010.

[11] Mário de Sousa. Ambiguities in IEC 61131-3 ST and IL expression semantics. In *13th IEEE International Conference on Industrial Informatics*, pages 1312–1317. IEEE, 2015.

[12] Borja Fernández Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Víctor M. González Suárez. Applying model checking to industrial-sized PLC programs. *IEEE Transactions on Industrial Informatics*, 11(6):1400–1410, 2015.

[13] Borja Fernández Adiego, Dániel Darvas, Jean-Charles Tournier, Enrique Blanco Viñuela, and Víctor M. González Suárez. Bringing automated model checking to PLC program development – A CERN case study. In Jean-Jacques Lesage, Jean-Marc Faure, José E. Ribiero Cury, and Bengt Lennartson, editors, *Proc. of the 12th International Workshop on Discrete Event Systems*, pages 394–399. IFAC, 2014.

[14] IEC 60848:2013 – GRAFCET specification language for sequential function charts, 2013.

[15] IEC 61131-3:2003 – Programmable controllers – Part 3: Programming languages, 2003.

[16] Ashutosh Kabra, Gopinath Karmakar, and R. K. Patil. A structured text to MISRA-C translator and issues with IEC 61131-3 standard. In *Proc. of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation*. IEEE, 2012.

[17] Maarten G. Meulen. Verification of PLC source code using propositional logic. Master's thesis, TU Eindhoven, 2010.

[18] Olivera Pavlović and Hans-Dieter Ehrich. Model checking PLC software written in function block diagram. In *Proc. of the International Conference on Software Testing, Verification and Validation*, pages 439–448. IEEE, 2010.

[19] Olivera Pavlović, Ralf Pinger, and Mail Kollmann. Automated formal verification of PLC programs written in IL. In Bernhard Beckert, editor, *Proc. of 4th International Verification Workshop*, volume 259 of *CEUR-WS*, pages 152–163. University of Koblenz-Landau, 2007.

[20] Jan Sadolewski. Automated conversion of ST control programs to Why for verification purposes. In *Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 849–854. IEEE, 2011.

[21] Jan Sadolewski. Conversion of ST control programs to ANSI C for verification purposes. *e-Informatica*, 5(1):65–76, 2011.

[22] Siemens. *SIMATIC Ladder Logic (LAD) for S7-300 and S7-400 Programming*, 1996. C79000-G7076-C504-02.

[23] Siemens. *SIMATIC Statement List (STL) for S7-300 and S7-400 Programming – Reference manual*, 1998. C79000-G7076-C565.

[24] Siemens. *SIMATIC Statement List (STL) for S7-300 and S7-400 Programming*, 2002. A5E00171232-01.

[25] Siemens. *S7-300 Instruction List*, 2010. A5E02354744-03.

[26] Siemens. *SIMATIC Programming with STEP 7*, 2010. A5E02789666-01.

[27] Siemens. *SIMATIC Safety – Configuring and Programming*, 2011. A5E02714440-01.

[28] Siemens. Standards compliance according to IEC 61131-3, 2011. `http://support.automation.siemens.com/WW/view/en/50204938`.

[29] André Sülflow and Rolf Drechsler. Verification of PLC programs using formal proof techniques. In Géza Tarnai and Eckehard Schnieder, editors, *Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 43–50. L'Harmattan, 2008.