Generic Representation of PLC Programming Languages for Formal Verification

Dániel Darvas*†, István Majzik* and Enrique Blanco Viñuela†
*Budapest University of Technology and Economics, Department of Measurement and Information Systems
Budapest, Hungary, Email: {darvas,majzik}@mit.bme.hu
†European Organization for Nuclear Research (CERN), Beams Department
Geneva, Switzerland, Email: {ddarvas,eblanco}@cern.ch

***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.**

I. Introduction and Background

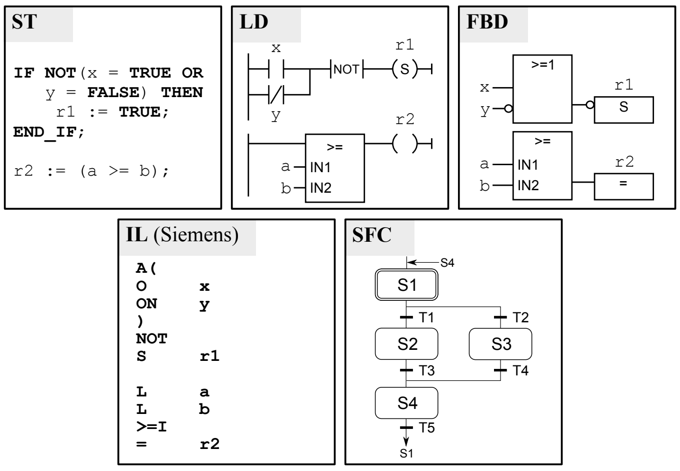
Programmable Logic Controllers (PLCs) are widely used for various control tasks in the industry. As they often perform critical tasks – sometimes PLCs are even used in safety-critical settings up to SIL3 – the verification of these hardware-software systems is a must. Besides the common testing and simulation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

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) [1]. It is out of the scope to discuss the features of these languages in detail, but a simple example in Figure 1 shows the different flavours of these languages. The first four example program excerpts are *execution equivalent*, i.e. for all 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.

A. Motivation

Our practical motivation lies in the PLCverif formal verification tool and its workflow [2], [3]. 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 formalized using verification patterns. At the moment, programs (or program



```
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
```

Fig. 1. PLC language examples

units) written in the Siemens variant of ST or SFC are supported. These inputs are convenient for the users 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 [4]. While most of the PLC programs are written in ST at CERN, in special cases (e.g. safety-critical applications) 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, the currently used Xtext-based 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 verification model. However, as it will be discussed later, the development and maintenance effort needed 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.

B. Related Work

Transformation of PLC programs were already studied previously. For example, Sadolewski translates ST programs into ANSI C [5] and Why [6] for verification purposes. However, in both work the source and target languages are on a similar abstraction level, not necessitating a detailed analysis of the ST language. Sülflow and Drechsler [7] translate IL programs to SystemC representation in order to perform equivalence checking. Here the translation involves a significant change in the abstraction level, requiring more considerations. Furthermore, all of them targeted one single source PLC language. In our work, all the five PLC languages are compared.

The paper is structured as follows. Section II defines our comparison method. Section III discusses the relations between the different IEC 61131 PLC programming languages. Next, Section IV discusses a concrete implementation of these languages, namely the one provided by Siemens. Section V analyzes the results of the paper and draws the conclusions. Finally, Section VI summarizes the paper.

II. 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 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. Firstly, because these languages are designed for special purposes, therefore they contain many limitations. One of them is the lack of dynamic memory allocation. Together with the lack of recursion [1, Sec. 2.5] and the limited data structures, it is impossible to use more storage than the amount defined explicitly in compilation time. These limitations are parts of the language definition, not caused by implementation or hardware limitations, therefore these languages are *not Turing complete*.

Secondly, 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"*[1] 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.

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

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 [1]. Later, we check the differences between the standard and the Siemens variants.

III. Standard Languages

In this section, we discuss the element-wise representation relation for each pair of standard PLC languages. The findings are summarized in Table I. Here "–" denotes that this representation is not possible.

TABLE I. Element-wise mapping between standard languages

from \ to	ST	IL	FBD	LD	SFC
ST	+	+	–	–	–
IL	–	+	–	–	–
FBD	–	+	+	+	–
LD	–	+	+	+	–
SFC	+	+	+	+	+

SFC is based on a specification method called Grafcet, 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. As only certain types of program units can be represented by SFCs, while the other four languages target all of the program unit types, no other language can be represented in SFC. Since it is based on Petri nets, translating the structure of an SFC program to any other language might be problematic, because Petri nets allow non-determinism, while the PLC languages are deterministic. However, determinism is explicitly required by the standard [1, Sec. 2.6.5]. The parts of SFC besides the structure are defined as simple program snippets in other languages and these specific parts can be easily mapped to any other PLC language, assuming that the ambiguities of the standard are first resolved [8].

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 is closer to the electric diagrams, with concepts like power rails, contacts and coils. Despite the differences, IEC 61131 defines LD and FBD in a similar way, with many common elements. The differences [1, 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, that is an assembly-like, low-level language. 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 [1, Sec. B.3]. Although it is known that Turing complete programs can be made jump-free by replacing jumps with loops and conditional statements [9], this construction does not fit to our approach of element-wise mapping.

IL has instructions such as `LD` (load value to accumulator) or `ST` (store the accumulator value to the given variable), and the other languages do not provide direct access to the accumulator, this way the element-wise (instruction by instruction) translation to any other PLC language is not possible.

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 allowed in ST, but not in FBD or LD [1, Sec. 2.4.1.2], therefore the ST to FBD or LD translation is not possible. On the other hand, these expressions are allowed in IL. More precisely, the ST syntax for defining expression is allowed in IL in certain cases. 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 above discussion and Table I, 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 [8], [10] and the vendors might resolve the ambiguities differently. The following section compares a concrete implementation of the five PLC programming languages.

IV. 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" feature or parameter 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, therefore we focus on the Siemens variants of the languages in this section. All five languages are supported in Siemens PLCs, however with some differences [11]. 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, 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.

The differences between the standard and Siemens versions of FBD, LD and SFC are subtle and mainly syntactic[2] [11].

[2]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 [11].

TABLE II. Element-wise mapping between Siemens languages

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

Notable differences in syntax and semantics between the standard and the implementation can be observed in the Siemens variants of ST and IL. The following part of the section overviews the differences compared to Table I, see Table II.

As the FBD', LD', and SFC' are equal to the standard versions, the relations between them are valid for the Siemens variants too. ST' and IL' are extended compared to the standard equivalents. Therefore if one of these languages can be mapped to ST or IL, it can be mapped also to the corresponding implementation, and if ST or IL cannot be mapped to one of these languages, IL' or ST' cannot be mapped to the implementation of the same language either. Consequently, the shaded cells of Table II are inherited from Table I.

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 [12], the translation from LD' and FBD' to IL' is always possible. We omit the discussion of transforming LD' and FBD' 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 III 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. Therefore 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 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 variant of **IL** is remarkably different from the standard IL. This is manifested in a different syntax. A short example is the following: the IL program in Listing 1 and the IL' program in Listing 2 give the same outputs to the same inputs, 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[3]. 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.

[3]From this point we use the term "register" in a generic way, referring to status bits, accumulators, nesting stack, etc.

```
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

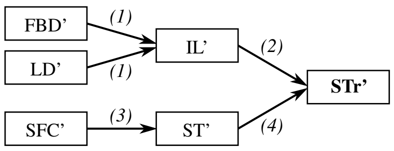


Fig. 2. Unified representation of Siemens PLC languages

V. Analysis and Conclusion

Looking at Table II might lead to the same conclusion for the Siemens implementations of the languages as Table I. However, due to the extensions in the implementations, the gap between IL' and ST' is much smaller than between their standard equivalents. The only 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 [7].

Although the FBD' and LD' 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 generator to construct the intermediate verification model fits all 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 summarizes 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 *(2)*. The SFC' to ST' translation is relatively simple, it can be implemented using the same principles as the ones used in [2] 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.

In this paper *interrupts* were not targeted. Certain PLCs may use interrupts, interrupting the execution of the main program. A certain IL' instruction can 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.

VI. 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 found: 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.

References

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

[2] B. Fernández *et al.*, "Applying model checking to industrial-sized PLC programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[3] D. Darvas, B. Fernández, and E. Blanco, "PLCverif: A tool to verify PLC programs based on model checking techniques," in *Proc. of the 15th Int. Conf. on Accelerator & Large Experimental Physics Control Systems*, 2015.

[4] B. Fernández, D. Darvas, J.-C. Tournier, E. Blanco, and V. M. González, "Bringing automated model checking to PLC program development – A CERN case study," in *Proc. of the 12th Int. Workshop on Discrete Event Systems*. IFAC, 2014, pp. 394–399.

[5] J. Sadolewski, "Conversion of ST control programs to ANSI C for verification purposes," *e-Informatica*, vol. 5, no. 1, pp. 65–76, 2011.

[6] ——, "Automated conversion of ST control programs to Why for verification purposes," in *Proc. of the Federated Conf. on Computer Science and Information Systems*. IEEE, 2011, pp. 849–854.

[7] A. Sülflow and R. Drechsler, "Verification of PLC programs using formal proof techniques," in *Formal Methods for Automation and Safety in Railway and Automotive Systems*. L'Harmattan, 2008, pp. 43–50.

[8] N. Bauer, R. Huuck, B. Lukoschus, and S. Engell, "A unifying semantics for sequential function charts," in *Integration of Software Specification Techniques for Applications in Engineering*, ser. Lecture Notes in Computer Science. Springer, 2004, vol. 3147, pp. 400–418.

[9] C. Böhm and G. Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules," *Communications of the ACM*, vol. 9, no. 5, pp. 366–371, 1966.

[10] M. de Sousa, "Proposed corrections to the IEC 61131-3 standard," *Computer Standards & Interfaces*, vol. 32, no. 5-6, pp. 312–320, 2010.

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

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