On Analysing the Semantics of IEC61131-3 ST and IL Applications

Mario de Sousa[1]

Electrical & Computer Engineering Department
University of Porto
4200-465 Porto, Portugal

ABSTRACT

The IEC 61508 standard recognizes the programming languages defined in IEC 61131-3 as being appropriate for safety-related applications, and suggests the use of static analysis techniques to find errors in the source code. In this context, we have added a semantic verification stage to the MatIEC compiler - an open source ST, IL and SFC code translator to ANSI C. In so doing, we have identified several issues related to the definition of the semantics of the IL and ST programming languages, as well as with the data type model defined in IEC 61131-3. Most of the issues are related to undefined semantics, which may result in applications generating distinct results, depending on the platform on which they are executed. In this paper we describe some of the issues we uncovered, explain the options we took, and suggest how the IEC 61131-3 standard could be made more explicit.

1. INTRODUCTION

Due to their robustness and flexibility, PLCs (Programmable Logic Controllers) have been used in many domains. However their use in high integrity and safety critical systems has often been conditioned on the requirement of having external equipment to monitor their correct behavior. This is mostly due to the difficulty in producing and verifying the correctness of their programs, since the hardware aspects have been mostly resolved through the use of hardware redundancy. The commercial availability of SIL 3 (SIL – Safety Integrity Level) rated PLCs on the market has not changed this, as what is SIL 3 rated is the hardware and the operating system of the PLC itself. In order to achieve the required SIL of the final system in which the SIL 3 PLC is integrated, then the program that is installed on the PLC also needs to be highly dependable. An example is a recently developed railway signaling that has been based on configurable PLC programs with the objective of being highly flexible and easily adaptable to the distinct railway lines configurations (the author worked as a consultant in this project).

In order to achieve this high reliability of the software, appropriate software development methods must be used. However, one of the main issues that affects the reliability of the final program is the programming language in which it is written. With the advent of the IEC 61131-3 standard, and its ever growing adoption by almost all PLC manufacturers, developing these highly dependable PLC programs is no longer out of the question. In fact, the IEC 61131-3 standard defines a common syntax and semantics for the programming languages commonly used for PLC programming. Although the programming languages defined in this standard are very similar to the programming languages previously used in programming PLCs, it was often the case the the languages used by distinct PLC manufacturers differed in some sometimes small but very important ways.

1.1. PLCS AND SAFETY-RELATED APPLICATIONS

Many previous works have already discussed how to design and/or restrict programming languages in order to make them useful for writing integrity applications. The IEC 61508 standard [1] (that defines the Functional Safety of Electrical, Electronic, and Programmable electronic Safety Related systems) already classifies common programming languages – interestingly it considers PLC programming languages as recommended for the development of high integrity systems, especially if these are somehow limited to a safe subset of the language. However, the standard itself does not specify the safe subset, nor does it reference any other document or standard where a subset of the IEC 61131-3 has been defined.

[1] Corresponding author: Tel.: (+351) 225081815; E-mail: msousa@fe.up.pt

Somewhat relatedly, the PLCOpen consortium has published a document [2] defining several Function Blocks that are useful in implementing safety related functions. This document also extends the elementary data types defined in IEC61131-3 – it defines an extra 'safe' version of each elementary data type, named SAFEINT, SAFEBOOL, etc. Apart from the extensions made to the base IEC 61131-3, this document also defines some restrictions to the IEC 61131-3 [3] languages and their development environments, with the intention of limiting the possibility of the use of unsafe programming practices that may result in such things as race conditions on preemption based systems.

One may consider that this would be a possible definition of the required subset of the IEC 66131-3 programming languages, however, what we have uncovered is that the IEC 61131-3 standard leaves some details of the specified languages with undefined semantics, and therefore one of two possible routes needs to be taken; either (a) the IEC 61131-3 standard needs to be modified and corrected so as to better specify the undefined behavior, or (b) a more restricted subset of the programming languages needs to be defined.

In this paper we have opted mostly for the first option (a). We have done this by suggesting changes to the standard that would be sufficient to correctly defined the as yet undefined semantics. Additional, we have validated the proposed semantics by extending an existing open source compiler (MatIEC [4]) of IEC 61131-3 languages in such a way as to implement the proposed semantics. Other changes made to the compiler add warnings to the user whenever the user makes use of language features that are not well defined in the original standard.

1.2. Previous work

Most previous work related to analysing the IEC 61131-3 standard has focused mostly on the syntax of the languages, and the ambiguities therein. These have been described by de Sousa [5] and Plaza et al. [6].

A lot of work has been done regarding the formalisation of programs written in the IEC 61131-3 languages, so they may be later formally verified (see [7] for a good overview of previous work in this area). However, this approach does not focus on analysing the IEC 61131-3 programming languages themselves.

At least two isolated groups have worked on formally defining the semantics of a sub-set of the IEC 61131-3 languages. Fett et al. [8] seem to have generated a compiler form this specification. However, in the literature it is not possible to find the details of the specification that was produced, nor a definition of the exact subset of the languages that was considered. On the other hand, Kourlas [9] focused on defining the semantics of only FBD (Function Block Diagrams), and criticise the standard for not defining their model of computation. Nevertheless, it is our view that this model is in fact defined, since the execution of the FBD is controlled by the POU (Program Organization Unit) in which it is declared, and the task in which the containing POU is executed.

1.3. Paper Organisation

After this introduction, this paper continues with a section 2 with an overview of the IEC 61131-3 data standard. In the following section 3 the detected semantic ambiguities are explained, while section 4 focuses on the MatIEC compiler and its overall architecture, with a focus on how the previously mentioned ambiguities where handled. Conclusions and outlook for future work is the object of the last section.

2. The IEC 61131-3 Standard

The IEC 61131-3 standard defines 4 programming languages (ST – Structured Text, IL – Instruction List, LD – Ladder Diagram, and FBD – Function Block Diagram), with an additional state based sequential programming model (i.e. SFC – Sequential Function Chart) which is often called a 5th programming language.

What is unique in this standard is that these 5 programming languages share the exact same data type model, as well as the same architectural or structural model. By structural model we mean the entities used to structure the code, namely the POUs (Program Organization Units): Functions, Function Blocks, Programs, and Configurations.

By sharing the same data type model, as well as the same structuring model, it becomes possible to write a single application using several distinct programming languages simultaneously, as long as each POU uses a single programming language.

2.1. Data Type Model

The IEC 61131-3 standard defines several elementary data types, which may be used to store unsigned integer values (USINT, UINT, UDINT, ULINT), signed integers (SINT, INT, DINT, LINT), real values (REAL, LREAL), boolean values (BOOL, BYTE, WORD, DWORD, LWORD), character strings (STRING, WSTRING), and time related values (TIME, TIME_OF_DAY, DATE_AND_TIME, and DATE). Note that a BYTE is only considered as a sequence of 8 boolean bits, with no inherent quantity involved, which clearly distinguishes it from a SINT.

The standard also specifies the use of strong data type consistency. This means that a variable of a specific data type can only take values of that same data type, and may therefore never be used as another data type. For example, a variable of type SINT cannot be used where a variable of data type INT is expected. This strong type consistency is good for checking program correctness, and appropriate for safety-related applications. However, it places many obstacles for the programmer who may need to use explicit type conversion functions (e.g. SINT_TO_INT) many times throughout their code.

In order to overcome this obstacle, the newer version of the IEC 61131-3 standard that is currently being drafted, will allow limited automatic type conversions (or type casts), and only where no information is lost. So, for example, a SINT may be used in place of an INT, but not the other way around. Since the current version of the standard does not support implicit type conversions, it may be considered as implementing a safe type system. Since the newer version also guarantees that that no information is lost in the implicit type conversions, it too will maintain the type safety of the type system.

The IEC 61131-3 standard also allows the user to define additional data types. These may be (a) derived from the elementary data types, or may (b) be constructed from complex structures based on these types. Examples of the first (a) data types, are simple renaming of an elementary data type, the changing of the default initial value, or the definition of a sub-range. For example:

```
TYPE
  analog_t: REAL;
  real1 : REAL := 1;
  current_t: USINT (4 .. 20);
END_TYPE
```

Examples of the second (b) are arrays, structures, and enumerations.

```
TYPE
  complex: STRUCT
              r: REAL;
              i: REAL;
           END_STRUCT;
  sample: ARRAY [-10 .. 0] of LREAL;
  colour: (black, brown, red, yellow);
END_TYPE
```

2.2. The Programming Languages

As previously stated, the IEC 61131-3 standard defines 4 programming languages. The two graphical languages, LD and FBD, are somewhat similar to designing an electrical circuit. With LD the circuit is based on series and parallel connections of relay contacts (representing reading of boolean variables) which energize the relay coils (writing to boolean variables). With FBD the electrical circuit is more akin to a digital circuit diagram using small scale integration integrated circuits (counters, timers, mutiplexors, etc.).

The two textual languages are very dissimilar. IL is comparable to assembly level, with several simple operations that only take one operand. The operation takes as parameters the operand and the value in an accumulator variable, with the result being in the same accumulator variable. However, this has been extended to allow the invocation of functions, and conditional jumps.

The ST textual language allows for a higher level of programming, as it is somewhat similar to the PASCAL programming language. It is a sequence of statements, including assignments, iterations (for, repeat, while) and

conditional execution (case, if/then/else). In all these statements, complex expressions may be used, that in turn may include function invocations.

These programming languages, especially the FBD and ST programming languages, were considered suitable for high integrity systems mainly due to their simplicity, and presumed fact that they were well defined.

3. Semantic Ambiguities

Several issues with the IEC 61131-3 standard have however already been raised in previous work, but these are mostly related to ambiguities or errors in the syntax. Here, we will point out ambiguities related to the semantics of the languages.

3.1 Semantic Ambiguity 1 - Data Type Equivalence

In terms of data type equivalences, the standard clearly states that data types that are directly derived from another base data type (e.g. renaming, or re-defining a new default initial value) are equivalent to the base data type. However, the standard is silent in relation to the remaining complex user defined data types, such as structures, arrays, and enumerations. In particular, it does not specify any data type equivalence rules for derived data types that are recursively defined based on other derived data types.

Common type equivalence rules for programming languages are the “structure equivalence”, and the “name equivalence”. In the first, data types with the same structure are considered equivalent, whatever the name used to identify the type and its sub-components, while in the second “name equivalence” the data types must have identical names.

In this case, and taking into account that the standard seems to want to define a strongly typed language, but nevertheless allows for data type equivalence as long as the base structure is not changed, we have previously proposed that the data type equivalence rules should be extended to allow something in between the two above mentioned rules[12]. In particular, derived data types that are directly derived from other complex derived data types shall be considered equivalent. However, derived data types that are newly defined are always considered distinct fromall other datatypes, even though they may have the same internal structure as another derived data type.

For example, consider the following data types:

```
TYPE
  c1 : STRUCT r: REAL; i: REAL; END_STRUCT;
  c2 : STRUCT r: REAL; i: REAL; END_STRUCT;
  c3 : STRUCT x: REAL; y: REAL; END_STRUCT;
  c2a: c2;
  c3a: c3;
END_TYPE
```

A programming language with structural equivalence data types will consider all the above data types as being equivalent. On the other hand, a name equivalence data type model will consider all the above data types distinct. With our proposed data type model for the IEC 61131-3, the data types c1, c2 and c3 are all distinct, while c2a and c2 are equivalent, as are c3and c3a.

3.2 Semantic Ambiguity 2 – Scope of Enumeration Identifiers

Identifiers in a programming language are used to identify specific program entities, for example: a data type name, a function name, a program name, etc. On the other hand, keywords are reserved identifiers that usually have a special meaning in the programming language, for example: Type, End_type, Array, Struct, etc. We have previously mentioned in [5] that the way IEC 61131-3 handles identifiers and reserved keywords is broken. However, we have recently realized that the use of identifiers related to the definition and use of enumerations is also ambiguous.

The issue here is that the standard allows the definition of derived data types that are an enumeration of several identifiers. For example:

```
TYPE
  colour_t: (black, brown, red, yellow);
  cable_t:  (black, red, brown);
END_TYPE
```

Additionally, variables may be defined to be of an anonymous data type.

```
Function foo : INT
  VAR product_colour: (black, white, gray); END_VAR
 ...
End_Function
```

In the above example, we have a derived data type 'colour_t', and a variable 'product_colour', is of an enumerated data type that dos not have a name, i.e. it is anonymous. Following the previously specified data type equivalence rules, the variable 'product_colour' is not compatible with variables of 'colour_t' data type. In particular, the enumeration constants used for the 'colour_t' are clearly distinct from the enumeration constants for the anonymous data type. This is also true for the 'black' enumeration constant, that shows up in both enumeration data types.

And Herein lies the issue. Although the standard clearly defines the scope of the 'colour_t' identifier as being global (more precisely, globally valid within the library in which it is defined), and the identifier 'product_colour' as being in scope merely within the function named 'foo', it does not specify the scope of the enumeration constants, although it does explicitly allow these to be re-used. If one is to assume that the enumeration constants in the globally defined data types also have a global scope, and the 'black' enumeration constant has local scope (local to 'foo'), then every reference to the 'black' enumeration constant is now ambiguous, since it may reference one of two (or possibly three) distinct data types.

For the globally valid named data types, the standard includes a special syntax that allows the enumeration constants to be disambiguated. For example:

```
colour_var := colour_t#black;
cable__var :=  cable_t#black;
```

However, the standard simply states that ' It is an error if sufficient information is not provided in an enumerated literal to determine its value unambiguously". The question arises if in the following case

```
colour_var := black;
cable__var := black;
```

the assignments are unambiguous. In reality, it is clearly possible to determine the referenced data type in the above case, from the data type of the variable in the left hand side of the assignment statement. However, since the standard states that the information must be present in the 'enumerated literal', which is not the case above, then one must conclude that the above assignments must be considered erroneous.

The issue arises when code inside the 'foo' function wishes to reference the 'black' enumeration constant belonging to the anonymous data type. This data type is anonymous, so it is not possible to disambiguate the 'black' constant using the above mentioned syntax. However, one may argue that the 'black' identifier, used inside the 'foo' function, hides the 'black' identifiers with global scope. In this case, the assignment (inside the 'foo function)

```
product_colour := black;
```

would have to be considered valid. Nevertheless, since the standard is silent on the issue of whether the locally scoped 'black' identifier hides the globally scoped 'black' identifiers, one may conclude that this is not the case. This has the implication that the locally scoped 'black' enumeration constant has no way of being referenced. The programmer would need to either (i) rename the locally scoped enumeration constant, or (ii) to define the data type as a non-anonymous data type.

The first option (i) is usually possible, as long as the programmer has access to all the source code. However, if he is importing the previously defined 'foo' function into his code, it does not make much sense that he be required to change the code of the 'foo' function so it may be used within his library, so it is not really an ideal solution. The second option (ii) is also not ideal, since derived data types may only be defined with a global scope, and may not

therefore be defined inside the 'foo' function itself, which is clearly what the intention of the above code (using the anonymous data type) would be.

Since neither of the above two solutions is ideal, we have opted to allow the hiding of the globally scoped 'black' enumeration constants in MatIEC, and therefore allow the following assignment within the 'foo' function.

```
product_colour := black;
```

However, in this case, we issue a warning clearly stating that this use of the 'black' identifier is dangerous and potentially ambiguous in other IEC 61131-3 development environments.

3.3 Semantic Ambiguity 3 – Evaluation Order of Function Invocation Parameters

The IEC 61131-3 standard allows functions to be called using any of the 4 programming languages. However, function invocation using either one of the textual languages allows for more complex situations to arise. In this case, the IEC 61131-3 textual languages allows ambiguities to arise due to the fact that the standard does not specify the order by which function invocation parameters are evaluated. For example (using the ST syntax):

```
var1 := foo( bar(in := 33, out => var2), var2);
```

In the above example, the 'foo' function is being called with 2 parameters being passed. The second parameter of the 'foo()' invocation is simply the value of the 'var2' variable. The first parameter is set to the result of another function invocation, namely 'bar()'. The 'bar()' function invocation also takes two parameters; the first is the constant '33', and the second is the variable 'var2'. However, in this case the variable 'var2' is being used to store the result of the output parameter 'out'.

Notice how the second parameter being passed to the 'foo()' function invocation will have a distinct value, depending on the order in which the two parameters are evaluated; is the value of 'var2' first passed to the second parameters of foo(), or is the bar() function invoked first?

The IEC 61131-3 standard leaves this order of function parameter evaluation completely unspecified (even though the order of evaluation of expressions with multiple operators is very clearly defined). This issue is not specific to the IEC 61131-3 standard, as this is also common in other widely used programming languages. One way of resolving this issue is to simply define a function parameter evaluation order, whichever it may be.

However, since this issue is common in other programming languages, and the evaluation order, if it exists, is usually not well known among software practitioners, it is much safer to simply avoid the use of the above situation. With this in mind, we have augmented the MatIEC compiler with code that is capable of detecting the above situations, even if they occur inside even more complex function invocations with multiply embedded invocations, and therefore issue a warning to the programmer.

The detection of this situation is based on an extension of the constant folding and constant propagation static code checking algorithms.

3. The MatIEC Compiler

As was stated previously, the MatIEC compiler is a code translator for ST, IL and SFC programs into ANSI C. It was originally developed for a now defunct project named MatPLC, which intended to produce an open-source PLC. It was later integrated into the Beremiz project (www.beremiz.org), which offers an integrated development environment (IDE) for developing IEC 61131-3 applications, including a graphical editor for SFC, LD and FBD programs.

Currently the MatIEC compiler is also being used by a few companies for their commercial products, either together with the Beremiz IDE, or with other proprietary IDEs. Typically, the IDEs convert graphical SFC programs to their textual representation, and FBD and LD programs to either IL or ST, which are all later compiled by the MatIEC compiler. The compiler itself is organized in four stages: lexical analyser, syntax parser, semantics analyser, and code generator.

The lexical parser analyses the source code and breaks it up into lexical tokens, removing on the way all comments and white-spaces between the tokens. The syntax parser groups the tokens into syntax constructs, and builds an equivalent internal abstract syntax tree data structure. The semantic analyser walks through the abstract

syntax tree and determines whether all semantic rules have been obeyed. The code generator, walks through the abstract syntax tree once again, and produces the final equivalent C code. This architecture allows us to easily write a new code generator for whatever output language desired, without having to rewrite all the lexical, syntactic and semantic parsers.

Our abstract syntax tree has been implemented as a tree of C++ objects, and the code that handles these objects follows the visitor design pattern [10]. This enables us to easily add or remove stages to our architecture without having to edit the abstract syntax tree classes themselves. Other future possible additions to the architecture include a code optimization stage.

3.1. Lexical Analyser

The lexical analyser was implemented using the flex utility that generates lexical analysers from a configuration file. The configuration file includes the extended expression definitions of the language's tokens.

This stage is the most straightforward, but is still nevertheless relatively complex due to its capability of parsing ST, Il and SFC code intermixed in the same input file. To do this we were required to use a state machine since not all languages have the same definition of tokens. For example, the EOL (end-of-line) token is considered white space in ST, but is relevant for parsing IL code.

3.2. Syntax Parser

The syntax parser was implemented using the GNU bison utility. This program generates a syntax parser from the syntax definition of the language being parsed. Although it too may have seemed straightforward at first, many issues had to be overcome but which are not the object of this paper.

3.3. The Semantic Checker

Due to a lack of time and resources, the semantic checker was left to a later stage, and only now has a first working version been implemented. The current version focuses mainly on data type consistency checking, but has already been extended to include constant folding and constant propagation.

Constant folding is the act of evaluating at compile time the result of every expression that has a constant value. For example, the code

```
x[42 + 9] := 99 / 3;
```

is replace with

```
x[51] := 33;
```

This allows us to better determine at compile time whether the ranges of arrays are being exceeded, or if the limits of a subrange variable is being exceeded. However, for situations with code

```
a := 9;
x[42 + a] := 99 / a;
```

the folding cannot be done since the expression contains a variable, even though the variable will always contain the same constant value in the code line being analysed. To cover these situations, we have implemented a constant propagation algorithm, that follows, for each location of the code where a variable is used, the possible values that the variable takes.

This algorithm is then expanded to take into account that the evaluation of function invocation parameters may be any, and then checks whether the results are always the same, whatever the evaluation order.

3.4. The Code Generators

IL and ST code transcription to C is rather straightforward as many of the constructs used in ST and IL are also available in C. Considering the POUs, Functions are mapped directly as C functions, while Function Blocks (FB) are mapped onto a structure that stores the FB's internal state (variables), as well as a C function containing the FB's code.

The resulting C code is practically self-contained and self-referencing, which allows it to be completely portable to any platform with a C compiler. The single instance that makes the code platform dependent is how located variables are mapped onto physical Input/Output.

The author is currently working on an intermediate code generator, so it may be integrated into the llvm family of compilers. This will allow for debugging features such as line by line code stepping, variable and parameter analysis, stack trace analysis, etc.

5. Conclusions

Although the IEC 61131-3 languages are suggested as being appropriate for high integrity applications, we have found that they are not however completely defined, with semantic ambiguities left undefined. To fix these ambiguities, we have either proposed a solution which has been validated by implementing it in the MatIEC compiler, or we have decided to leave them undefined, but have created a check that warns programmers that their programs relies on undefined behavior.

However, the MatIEC does not yet do a complete semantic analysis of all source code. In the future we intend to continue to implement more checks of semantic correctness, as well as augment the MatIEC compiler with coding style verifications, which is common in high integrity applications..

References

[1] *International Electrotechnical Commission, "Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems", December 1998.*

[2] *PLCopen - Technical Committee 5, "Safety Software Technical Specification, Part 1: Concepts and Function Blocks", Version 1.0, January 2006.*

[3] *International Electrotechnical Commission, "International Standard IEC 61131-3, Programmable Logic Controllers Part 3", Geneva, 1993.*

[4] *Mário de Sousa, Edouard Tisserant, Laurent Bessard, "An open source IEC 61131-3 integrated development environment", in 5th IEEE International Conference on Industrial Informatics 2007 (IEEE INDIN'07), pp.183-187, 2007*

[5] *Mário de Sousa, "Proposed corrections to the IEC 61131-3 standard", in Computer Standards and Interfaces, 32:312–320, October 2010.*

[6] *I. Plaza, C. Medrano, and A. Blesa, "Analysis and implementation of the IEC 61131-3 software model under POSIX real-time operating systems", in Microprocessor and Microsystems, 30:497–508, December 2006.*

[7] *M. Bani Younis and G. Frey, "Formalization of existing PLC Programs: A Survey", in in Proc. CESA, Lille (France) CD-ROM. Paper S2-R-00-0239, July 2003*

[8] *G. Egger, A. Fett, P. Pepper, "Formal Specification of a Safe PLC Language and its Compiler", in SAFECOMP'94, Proceedings of the 13th International Conference on Computer Safety, Reliability and Security, Anaheim, Kalifornien, USA*

[9] *K. Tourlas, "An Assessment of the IEC 61131-3 Standard Languages for Programmable Controllers", in SAFECOMP'97, Proceedings of the 16th International Conference on Computer Safety, Reliability and Security, York, United Kingdom, September 1997.*

[10] *"Design Patterns for Flexible Manufacturing", Dennis Brandl, ISA-Instrumentation, Systems, and Automation Society, 2007, ISBN-13: 978-1-55617-998-3.*

[11] *Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools", 2nd Edition, September 10, 2006, ISBN-13: 978-0321486813*

[12] *Mário de Sousa, "Data-Type Checking of IEC 61131-3 IL and ST Programs", in 17th IEEE International Conference on Emerging Technologies and Factory Automation (IEEE ETFA'12), 2012*