A syntactic specification for the programming languages of the IEC 61131-3 standard

Article · January 2010

CITATION
1

READS
4,680

3 authors, including:

Flor Narciso
University of Puerto Rico at Mayagüez
10 PUBLICATIONS **56** CITATIONS
SEE PROFILE

Francisco Hidrobo
University of the Andes (Venezuela)
62 PUBLICATIONS **241** CITATIONS
SEE PROFILE

A Syntactic Specification for the Programming Languages of the IEC 61131-3 Standard

FLOR NARCISO[§] ADDISON RIOS-BOLIVAR[§] FRANCISCO HIDROBO[‡]
OLGA GONZALEZ[§]

[§]Universidad de Los Andes
Escuela de Ingeniería de Sistemas
La Hechicera, Mérida 5101
VENEZUELA
fnarciso,ilich,ogonzalez@ula.ve

[‡]Universidad de Los Andes
Departamento de Física
La Hechicera, Mérida 5101
VENEZUELA
hidrobo@ula.ve

Abstract: In order to implement a translator, a syntactic specification for the programming languages of the IEC 611131-3 standard is presented in this paper. This specification is built in function of the components of a formal grammar, which are the set of terminal symbols, the set of non terminal symbols, the start symbol or axiom and the set of production rules. The formal grammar described corresponds to a context-free grammar; which is currently being used to describe most of the programming languages. With the definition of this grammar, the programming languages of the IEC 611131-3 standard are formally described and is presented as an example, the syntactic specification of the textual programming languages, known as Instruction List and Structured Text. Then, the graphical programming languages (Ladder Diagram and Functional Block Diagram) can be specified in the same way. With the grammar described is possible to build a translator using open standards and open source that can generate code in high level programming language (like C programming language) to support the management of control logics.

Key–Words: Context-Free Grammar, Programming Languages, IEC 611131-3 standard, Industrial Automation, Translator.

1 Introduction

A Programmable Logic Controller (PLC) is an electronic system of digital operation, which is designed to be used on industrial environments. This system uses a programmable memory for storage of the users instructions, these instructions are implemented through specific functions as logical, sequential, temporal, counters and arithmetic, to control, using analogical or digital input/output, diverse types of machines or processes [8]. The PLCs are well known systems in the industrial field, and they are widely used, along with personal computers (PCs) in control applications, conducing to the development of applications that allow a PC to function as a PLC. This kind of applications are called SoftPLC. Then, a SoftPLC is an application that runs in real time on a PC allowing to emulate the behavior of a PLC by software.

In order to assist control engineers in the development of control applications, using one or several standard programming language of PLC, and fulfilling with the premises of open source and the IEC 611131-3 standard [1], in [5, 7] the design of a computational tool for managing the control logics is described. The IEC 611131-3 standard aims to present an industrial standardization of programables automatas and their peripherals, including the programming languages to be used [2]. This standard provides an industrial standardization of programmable automatas and their peripherals, including the programming languages to be used: textual or literals programming languages (Instructions List (IL) and Structured Text (ST)) and graphical programming languages (Ladder Diagram (LD) and Functional Block Diagram (FDB)) [2].

The main objective of this computational tool, called open source SoftPLC [5, 7], is to offer a development environment for control applications that provides the programming languages of the IEC 611131-3 standard, for PLC programmers and engineers to build applications using the services and resources that form part of this tool. These services and resources, in their basic form, are joined in components well defined, such as an editor, a translator and a graphical user interface. This draws a substantial difference with respect to similar commercial tools, because the proposed tool meets all requirements of the standard and it is open source.

As stated earlier, one of the components of the proposed computational tool is a translator, which is a software application that reads a program written in a language, called the source language, and translates it into an equivalent program written in another language, called the target language or object language. As an important part of the translation process, the translator informs to the user the presence of errors, if any, in the program written in the source language [3]. In the particular case of this tool, the translator must transform a program written in any of

the programming languages provided by the IEC 611131-3 standard to an equivalent program written in an intermediate language. In this case the intermediate language under consideration is the C programming language [9], then obtained as a result of the translation process a code in C programming language that can be compiled and executed.

Conceptually, a translator works in phases, each of which transforms the program written in source language (source program) from one representation to another to get the program written in intermediate language (object program). Figure 1 shows a typical structure of a translator as the one proposed in [7] where the two functions, error management and administration of the symbol table are continuously interacting with the analysis and synthesis phases.

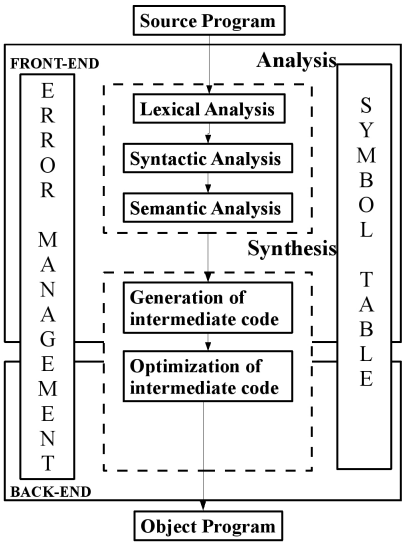

Figure 1: The graphical representation of the translation process

To make the syntactic specification of a programming language the Context-Free Grammars (CFG) are used, which allow to describe most of the programming languages [3]. Generally, these grammars are simple, allowing the design of syntactic-analysis algorithms which can determine whether a sequence of tokens correspond to a valid sentence of the programming language. Also, the CFG can be used in the design of lexical analyzers contributing to the identification and structuring of the collection of lexical components or tokens. In this paper, the syntactic specification of the PLC programming languages is presented, according to the established in the IEC 611131-3 standard, based on the description presented in [5] for the common elements of these programming languages (textual and graphicals). Following, the syntactic specification of the IL programming language is presented. Finally, the syntactic specification of the ST programming language is presented. The syntactic specification of the LD and FDB programming languages can be reviewed in [5].

2 Formal Grammar and Language

In [4], a language, from the linguistic point of view, is defined as "a set (finite or infinite) of sentences, each one of them with finite length and constructed with a finite number of elements". This definition can be used both for natural language (specific to humans) and for programming languages (specific to computers). A language formally specified is called formal language. The basic concept in formal language theory is the word, which consists of an ordered sequence of symbols from a specific alphabet, being the alphabet, denoted by $\sum$, the set of all symbols that form the language words. On the other hand, a set of words form a a sentence of the language.

In order to make the definition of the sentences of a language the grammar are used. These allow to make the syntactic specification of the programming languages, taking in account that these languages, like natural languages, are formed by a set of valid words (vocabulary), a set of grammatical rules that allow the construction of valid sentences of the language (syntax) and a set of rules that determine the meaning of the sentences of the language (semantics). The term formal grammar refers to the grammar that is defined in the computational linguistic. A formal grammar is formally defined as the ordered quadruple *G = (N, T, S, P)* [4]. The set of non-terminal symbols (N) is the finite set of symbols introduced as auxiliary elements for definition of the production rules of the grammar, none of which appear in the words of language. The set of terminal symbols (N) is the set of atomic symbols (indivisible) of the alphabet $\sum$, that are combined to form the words of language. The start symbol (S) is a non-terminal symbol from which all sentences of the language generated by the grammar are obtained by applying its rules of production. The set of production rules (P) are transformations of strings of symbols that can be applied successively from the start symbol to get a string of terminals that form a sentence of the language. To represent a production rule the notation $\alpha \rightarrow \beta$ is used.

The syntax of natural languages, programming languages and other formal languages are defined using context-free grammars (CFG) [3], in which the production rules of P have the form $A \rightarrow \beta$, such that, $A \in N$ and $\beta \in (N \cup T)^*$, where $(N \cup T)^*$ represent the set of all possible words that can be formed with the symbols of $(N \cup T)$ including the empty word [3]. Generally, the EBNF (Extended Backus Naur Form) notation is used to specify the production rules of a CFG, which is an extension of BNF (Backus Naur Form) notation, and it is defined as metalanguage that is used to express the syntax of programming languages. This notation uses the symbols shown in the table 1.

In order to obtain a syntactic specification of the programming languages of the IEC 611131-3 standard, in the section 3 it is described a CFG whose production rules are specified using the EBNF notation.

Table 1: EBNF notation symbols

Symbol	Description
::=	It separates the left and right part of the production rule and it is read: "it defined as"
|	It separates the alternatives of the right part of the production rules and it is read "or"
{}	Repetition, it means zero or more concatenations of items enclosed
[ ]	Option, it means zero or one occurrence of the items enclosed

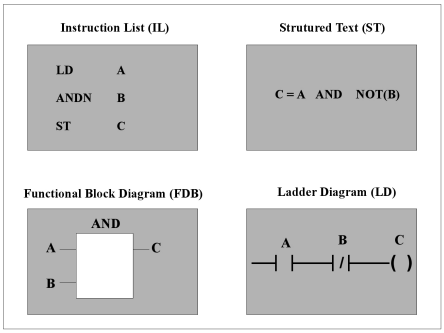

Figure 2: Example of a program written in the programming languages of the IEC 611131-3 standard

3 Syntactic specification of the textual programming languages for PLC

In this section, a syntactic specification for the common parts of the programming languages for PLC (textual and graphicals) as well as those specific for the textual programming languages for PLC (IL and ST) is presented, in terms of a CFG. The syntactic specification for the graphical programming languages for PLC (LD and FBD) have been described in [5], which completes the definition of the CFG that generates the four programming languages provided by the IEC 61131-3 standard. IL is a low-level programming language (machine dependent) that is similar to an assembly language for simple applications. ST is a high-level programing language (machine independent) that is similar to Pascal. LD is based on the graphical representation of the relay logic and it allow to describe an input logical sequence of a specific process using connections, contacts and coils. FBD is very common in applications involving information flows or data among control components, and it uses the functional block form the area of control process, which are combined in a logic by mean of signals. Figure 2 shows an example of a program written in each of the programming languages for PLC. In addition,the IEC 61131-3 standar provides a fifth programming language known as Sequential Function Chart (SFC), which allows to develop programs of sequential control, graphically, and its elements can be used in conjunction with any of the graphical programming languages described before.

In order to make the syntactic specification, the guidelines established in the IEC 611131-3 standard are followed, where the terminal symbols consist of a string enclosed in single or double quotes, for example, "ABC" or 'ABC'. A special terminal symbol used in this syntax is the end-of-line delimiter, which is represented by the string without quotes **EOL**. Normally, this symbol consists of the character "paragraph separator" defined as the hexadecimal code 2029 by the ISO/IEC 10646-1. A second special symbol used in this specification is the "NULL string", which have no characters, and it is represents by the terminal symbol **NIL** [2].

As it is defined in the IEC 611131-3 standard, the non-terminal symbols are represented by strings of lowercase letters, numbers and the underscore (_) character starting with a lowercase letter. For example, **term1** and **term_2** are valid non-terminal symbols, while **3noterm** and **Noterm4** are invalid non-terminal symbols [2]. The start symbol corresponds to the non-terminal symbol **library_element_name**.

The production rules for textual programming languages of PLCs have the form *Non-terminal_symbol ::=extended_structure*. This production rule can be read as "a non-terminal symbol is defined as an extended structure". The extended structures can be constructed according to the following criteria:

1. The NULL string, NIL, is an extended structure.
2. A terminal symbol is an extended structure.
3. A non-terminal symbol is an extended structure.
4. If S is an extended structure, then these expressions also are extended structures: (S), meaning the same S; $\{S\}$, repetition, meaning zero or more concatenations of S; $[S]$, option, meaning zero or one occurrence of S.
5. If $S1$ and $S2$ are extended structures, then these expressions are extended structures: $S1|S2$, alternative, meaning an option of $S1$ or $S2$; $S1\,S2$, concatenation, meaning $S1$ followed by $S2$.
6. The concatenation precede to alternative, $S1|S2\ S3$ is equivalent to $S1|(S2\ S3)$, and $S1\ S2|S3$ is equivalent to $(S1\ S2)|S3$.

In next section, the production rules that describe the common elements of the programming languages of the IEC 611131-3 standard are presented. These production rules represent the fundamental base for the design and implementation of a translator that allow the generation of high-level code from a program written in any of programming languages provided by this standard.

ISBN: 978-960-474-257-8

3.1 Production rules for the common elements of the programming languages for PLC

A *programming model* is defined according to the following production rules:

```
library_element_name ::= data_type_name | function_name | function_block_type_name |
program_type_name | resource_type_name | configuration_name
library_element_declaration ::= data_type_declaration | function_declaration |
function_block_declaration | program_declaration | configuration_declaration
```

These production rules express the basic programming model defined in [6], where declarations are the basic mechanism for production of the *library elements*. Next, the production rules for the common elements of the programming languages of the IEC 611131-3 standard are described, which correspond to *characters*, *digits* and *identifiers*; *constants*; *data types*; *variables*; *units of programs organization*; and *configuration elements*:

```
letter ::= 'A' | 'B' | <...> | 'Z' | 'a' | 'b' | <...> | 'z'
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |'9'
octal_digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
hex_digit ::= digit | 'A'| 'B' | 'C' | 'D' | 'E' | 'F'
identifier ::= (letter | ('_' (letter | digit))) {['_'] (letter | digit)}


constant ::= numeric_literal | character_string | time_literal | bit_string_literal |
boolean_literal

numeric_literal ::= integer_literal | real_literal
integer_literal ::= [ integer_type_name '#' ]( signed_integer | binary_integer |
octal_integer | hex_integer)
signed_integer ::= ['+' |'-'] integer
integer ::= digit {['_'] digit}
binary_integer ::= '2#' bit {['_'] bit}
bit ::= '1' | '0'
octal_integer ::= '8#' octal_digit {['_'] octal_digit}
hex_integer ::= '16#' hex_digit{['_'] hex_digit}
real_literal ::= [ real_type_name '#' ]
signed_integer '.' integer [exponent]
exponent ::= ('E' | 'e')['+'|'-'] integer
bit_string_literal ::= [ ('BYTE' | 'WORD' | 'DWORD' | 'LWORD') '#' ]( unsigned_integer |
binary_integer | octal_integer | hex_integer)
boolean_literal ::= ( [ 'BOOL#' ] ('1' | '0' ) )| 'TRUE' | 'FALSE'

character_string ::= single_byte_character_string | double_byte_character_string
single_byte_character_string ::= "'" {single_byte_character_representation} "'"
double_byte_character_string ::= '"' {double_byte_character_representation} '"'
single_byte_character_representation ::= common_character_representation | "$'" | '"' |
'$' hex_digit hex_digit
double_byte_character_representation ::= common_character_representation | '$"' | "'"|
'$' hex_digit hex_digit hex_digit hex_digit
common_character_representation ::= <any printable character except '$', '"' or "'"> |
'$$' | '$L' | '$N' | '$P' | '$R' | '$T' | '$l' | '$n' | '$p' | '$r' | '$t'

time_literal ::= duration | time_of_day
duration ::= ('T' | 'TIME') '#' ['-'] interval
interval ::= days | hours | minutes | seconds | milliseconds
days ::= fixed_point ('d') | integer ('d') ['_'] hours
fixed_point ::= integer [ '.' integer]
hours ::= fixed_point ('h') | integer ('h')['_'] minutes
minutes ::= fixed_point ('m') | integer ('m') ['_'] seconds
seconds ::= fixed_point ('s') | integer ('s') ['_']milliseconds
milliseconds ::= fixed_point ('ms')

time_of_day ::= ('TIME_OF_DAY' | 'TOD') '#' daytime
daytime ::= day_hour ':' day_minute ':' day_second
day_hour ::= integer
day_minute ::= integer
day_second ::= fixed_point
date ::= ('DATE' | 'D') '#' date_literal
date_literal ::= year '-' month '-' day
year ::= integer
month ::= integer
day ::= integer
date_and_time ::= ('DATE_AND_TIME' | 'DT') '#' date_literal '-' daytime


data_type_name ::= non_generic_type_name | generic_type_name

non_generic_type_name ::= elementary_type_name | derived_type_name

elementary_type_name ::= numeric_type_name | date_type_name | bit_string_type_name
| 'STRING' | 'WSTRING' | 'TIME'
numeric_type_name ::= integer_type_name | real_type_name
integer_type_name ::= signed_integer_type_name | unsigned_integer_type_name
signed_integer_type_name ::= 'SINT' | 'INT' | 'DINT' | 'LINT'
unsigned_integer_type_name ::= 'USINT' | 'UINT' | 'UDINT' | 'ULINT'
real_type_name ::= 'REAL' | 'LREAL'
date_type_name ::= 'DATE' | 'TIME_OF_DAY' | 'TOD' | 'DATE_AND_TIME' | 'DT'
bit_string_type_name ::= 'BOOL' | 'BYTE' | 'WORD' | 'DWORD' | 'LWORD'

generic_type_name ::= 'ANY' | 'ANY_DERIVED' | 'ANY_ELEMENTARY' | 'ANY_MAGNITUDE'
| 'ANY_NUM' | 'ANY_REAL' | 'ANY_INT' | 'ANY_BIT' | 'ANY_STRING' | 'ANY_DATE'
derived_type_name ::= single_element_type_name | array_type_name | structure_type_name |
string_type_name
single_element_type_name ::= simple_type_name | subrange_type_name | enumerated_type_name

simple_type_name ::= identifier
subrange_type_name ::= identifier
enumerated_type_name ::= identifier
array_type_name ::= identifier
structure_type_name ::= identifier
data_type_declaration ::= 'TYPE' type_declaration ';' {type_declaration ';'} 'END_TYPE'
type_declaration ::= single_element_type_declaration | array_type_declaration |
structure_type_declaration | string_type_declaration
single_element_type_declaration ::= simple_type_declaration | subrange_type_declaration |
enumerated_type_declaration
simple_type_declaration ::= simple_type_name ':' simple_spec_init
simple_spec_init ::= simple_specification [':=' constant]
simple_specification ::= elementary_type_name | simple_type_name
subrange_type_declaration ::= subrange_type_name ':'  subrange_spec_init
subrange_spec_init ::= subrange_specification[':=' signed_integer]
subrange_specification ::= integer_type_name '(' subrange')' | subrange_type_name
subrange ::= signed_integer '..' signed_integer
enumerated_type_declaration ::= enumerated_type_name ':'  numerated_spec_init
enumerated_spec_init ::= enumerated_specification [':=' numerated_value]
enumerated_specification ::= ('(' enumerated_value {',' enumerated_value} ')') |
enumerated_type_name
enumerated_value ::= [enumerated_type_name '#'] identifier
array_type_declaration ::= array_type_name ':' array_spec_init
array_spec_init ::= array_specification [':=' array_initialization]
array_specification ::= array_type_name | 'ARRAY' '[' subrange {',' subrange} ']'
'OF' non_generic_type_name
array_initialization::= '[' array_initial_elements {',' array_initial_elements} ']'
array_initial_elements ::= array_initial_element | integer '('[array_initial_element] ')'
array_initial_element ::= constant | enumerated_value | structure_initialization |
array_initialization
structure_type_declaration ::= structure_type_name ':' structure_specification
structure_specification ::= structure_declaration | initialized_structure
initialized_structure ::=structure_type_name [':=' structure_initialization]
structure_declaration ::='STRUCT' structure_element_declaration ';
'{structure_element_declaration ';'}'END_STRUCT'
structure_element_declaration ::= structure_element_name ':'(simple_spec_init |
subrange_spec_init | enumerated_spec_init | array_spec_init | initialized_structure)
structure_element_name ::= identifier
structure_initialization ::= '(' structure_element_initialization
{',' structure_element_initialization} ')'
structure_element_initialization ::= structure_element_name ':=' (constant | enumerated_value
| array_initialization | structure_initialization)
string_type_name ::= identifier
string_type_declaration ::= string_type_name ':' ('STRING'|'WSTRING') ['[' integer ']']
[':=' character_string]


variable ::= direct_variable | symbolic_variable
symbolic_variable ::= variable_name | multi_element_variable
variable_name ::= identifier

direct_variable ::= '%' location_prefix size_prefix integer {'.' integer}
location_prefix ::= 'I' | 'Q' | 'M'
size_prefix ::= NIL | 'X' | 'B' | 'W' | 'D' | 'L'

multi_element_variable ::= array_variable | structured_variable
array_variable ::= subscripted_variable subscript_list
subscripted_variable ::= symbolic_variable
subscript_list ::= '[' subscript {',' subscript} ']'
subscript ::= expression
structured_variable ::= record_variable '.' field_selector
record_variable ::= symbolic_variable
field_selector ::= identifier

input_declarations ::= 'VAR_INPUT' ['RETAIN' | 'NON_RETAIN'] input_declaration ';'
     {input_declaration ';'} 'END_VAR'
input_declaration ::= var_init_decl | edge_declaration
edge_declaration ::= var1_list ':' 'BOOL' ('R_EDGE' | 'F_EDGE')
var_init_decl ::= var1_init_decl | array_var_init_decl | structured_var_init_decl
| fb_name_decl | string_var_declaration
var1_init_decl ::= var1_list ':' (simple_spec_init | subrange_spec_init |
     enumerated_spec_init)
var1_list ::= variable_name {',' variable_name}
array_var_init_decl ::= var1_list ':' array_spec_init
structured_var_init_decl ::= var1_list ':' initialized_structure
fb_name_decl ::= fb_name_list ':' function_block_type_name [ ':=' structure_initialization ]
fb_name_list ::= fb_name {',' fb_name}
fb_name ::= identifier
output_declarations ::= 'VAR_OUTPUT' ['RETAIN' | 'NON_RETAIN'] var_init_decl ';'
     {var_init_decl ';'} 'END_VAR'
input_output_declarations ::= 'VAR_IN_OUT' var_declaration ';' {var_declaration ';'} 'END_VAR'
var_declaration ::= temp_var_decl | fb_name_decl
temp_var_decl ::= var1_declaration | array_var_declaration | structured_var_declaration |
     string_var_declaration
var1_declaration ::= var1_list ':' (simple_specification | subrange_specification |
     enumerated_specification)
array_var_declaration ::= var1_list ':' array_specification
structured_var_declaration ::= var1_list ':' structure_type_name
var_declarations ::= 'VAR' ['CONSTANT'] var_init_decl ';' {(var_init_decl ';')}
     'END_VAR'
retentive_var_declarations ::= 'VAR' 'RETAIN' var_init_decl ';' {var_init_decl ';'}
     'END_VAR'
located_var_declarations ::= 'VAR' ['CONSTANT' | 'RETAIN' | 'NON_RETAIN']
     located_var_decl ';' {located_var_decl ';'} 'END_VAR'
located_var_decl ::= [variable_name] location ':' located_var_spec_init
external_var_declarations ::= 'VAR_EXTERNAL' ['CONSTANT']
     external_declaration ';' {external_declaration ';'} 'END_VAR'
external_declaration ::= global_var_name ':' (simple_specification |
     subrange_specification | enumerated_specification | array_specification |
     structure_type_name | function_block_type_name)
global_var_name ::= identifier
global_var_declarations ::= 'VAR_GLOBAL' ['CONSTANT' | 'RETAIN']
     global_var_decl ';' {global_var_decl ';'} 'END_VAR'
global_var_decl ::= global_var_spec ':'
     [ located_var_spec_init | function_block_type_name ]
global_var_spec ::= global_var_list | [global_var_name] location
     located_var_spec_init ::= simple_spec_init | subrange_spec_init
     | enumerated_spec_init | array_spec_init | initialized_structure |
     single_byte_string_spec | double_byte_string_spec
location ::= 'AT' direct_variable
global_var_list ::= global_var_name {',' global_var_name}
string_var_declaration ::= single_byte_string_var_declaration
     | double_byte_string_var_declaration
single_byte_string_var_declaration ::= var1_list ':' single_byte_string_spec
```

```
single_byte_string_spec ::= 'STRING' ['[' integer ']'] [':=' single_byte_character_string]
double_byte_string_var_declaration ::= varl_list ':' double_byte_string_spec
double_byte_string_spec ::= 'WSTRING' ['[' integer ']'] [':=' double_byte_character_string]
incompl_located_var_declarations ::= 'VAR' ['RETAIN'|'NON_RETAIN']
     incompl_located_var_decl ';' {incompl_located_var_decl ';'} 'END_VAR'
incompl_located_var_decl ::= variable_name incompl_location ':' var_spec
incompl_location ::= 'AT' '%' ('I' | 'Q' | 'M') '*'
var_spec ::= simple_specification | subrange_specification | enumerated_specification |
     array_specification | structure_type_name | 'STRING' ['[' integer ']'] | 'WSTRING' ['[' integer ']']


function_name ::= standard_function_name | derived_function_name
standard_function_name ::= <as defined in clause 2.5.1.5 of the standard>
derived_function_name ::= identifier
function_declaration ::= 'FUNCTION' derived_function_name ':'
     (elementary_type_name | derived_type_name)
     { io_var_declarations | function_var_decls } function_body
     'END_FUNCTION'
io_var_declarations ::= input_declarations | output_declarations |
     input_output_declarations
function_var_decls ::= 'VAR' ['CONSTANT']
     var2_init_decl ';' {var2_init_decl ';'} 'END_VAR'
function_body ::= ladder_diagram | function_block_diagram |
     instruction_list | statement_list | <other languages>
var2_init_decl ::= varl_init_decl | array_var_init_decl |
     structured_var_init_decl | string_var_declaration
```

This syntax do not include the fact each function must have at least one input declaration. Also, It does not include either the fact that declarations of edge, references and invocations to function blocks are not allowed in the body of functions according to the standard [2].

```
function_block_type_name ::= standard_function_block_name
| derived_function_block_name
standard_function_block_name ::= <as defined in clause 2.5.2.3 of the
standard>
derived_function_block_name ::= identifier
function_block_declaration ::=
'FUNCTION_BLOCK' derived_function_block_name
{ io_var_declarations | other_var_declarations }
function_block_body
'END_FUNCTION_BLOCK'
other_var_declarations ::= external_var_declarations | var_declarations
| retentive_var_declarations | non_retentive_var_declarations
| temp_var_decls | incompl_located_var_declarations
temp_var_decls ::=
'VAR_TEMP'
temp_var_decl ';'
{temp_var_decl ';'}
'END_VAR'
non_retentive_var_decls ::=
'VAR' 'NON_RETAIN'
var_init_decl ';'
{var_init_decl ';'}
'END_VAR'
function_block_body ::= ladder_diagram | function_block_diagram
| instruction_list | statement_list | <other languages>

program_type_name :: = identifier
program_declaration ::=
'PROGRAM' program_type_name
{ io_var_declarations | other_var_declarations
| located_var_declarations | program_access_decls }
function_block_body
'END_PROGRAM'
program_access_decls ::=
'VAR_ACCESS' program_access_decl ';'
{program_access_decl ';' }
'END_VAR'
program_access_decl ::= access_name ':' symbolic_variable ':'
non_generic_type_name [direction]


configuration_name ::= identifier
resource_type_name ::= identifier
configuration_declaration ::=
'CONFIGURATION' configuration_name
[global_var_declarations]
(single_resource_declaration
| (resource_declaration {resource_declaration}))
[access_declarations]
[instance_specific_initializations]
'END_CONFIGURATION'
resource_declaration ::=
'RESOURCE' resource_name 'ON' resource_type_name
[global_var_declarations]
single_resource_declaration
'END_RESOURCE'
single_resource_declaration ::=
{task_configuration ';'}
program_configuration ';'
{program_configuration ';'}
resource_name ::= identifier
access_declarations ::=
'VAR_ACCESS'
access_declaration ';'
{access_declaration ';'}
'END_VAR'
access_declaration ::= access_name ':' access_path ':'
non_generic_type_name [direction]
access_path ::= [resource_name '.'] direct_variable
| [resource_name '.'] [program_name '.']
{fb_name'.'} symbolic_variable
global_var_reference ::=
[resource_name '.'] global_var_name ['.' structure_element_name]
access_name ::= identifier
program_output_reference ::= program_name '.' symbolic_variable
program_name ::= identifier
direction ::= 'READ_WRITE' | 'READ_ONLY'
task_configuration ::= 'TASK' task_name task_initialization
task_name := identifier
task_initialization ::=
'(' ['SINGLE' ':=' data_source ','
['INTERVAL' ':=' data_source ','
'PRIORITY' ':=' integer ')'
data_source ::= constant | global_var_reference
| program_output_reference | direct_variable
program_configuration ::=
'PROGRAM' [RETAIN | NON_RETAIN]
program_name ['WITH' task_name] ':' program_type_name
['(' prog_conf_elements ')']
prog_conf_elements ::= prog_conf_element {',' prog_conf_element}
prog_conf_element ::= fb_task | prog_cnxn
fb_task ::= fb_name 'WITH' task_name
prog_cnxn ::= symbolic_variable ':=' prog_data_source
| symbolic_variable '=>' data_sink
prog_data_source ::= constant | enumerated_value | global_var_reference |
direct_variable
data_sink ::= global_var_reference | direct_variable
instance_specific_initializations ::=
'VAR_CONFIG'
instance_specific_init ';'
{instance_specific_init ';'}
'END_VAR'
instance_specific_init ::=
resource_name '.' program_name '.' {fb_name '.'}
((variable_name [location] ':' located_var_spec_init) |
(fb_name ':' function_block_type_name ':='
structure_initialization))
```

3.2 Production rules for the elements of the IL programming language

In this section, the production rules that describe the elements of the IL programming language are presented, which correspond to *instructions*, *operands* and *operators*.

```
instruction_list ::= il_instruction {il_instruction}
il_instruction ::= [label':'] [ il_simple_operation
| il_expression
| il_jump_operation
| il_fb_call
| il_formal_funct_call
| il_return_operator ] EOL {EOL}
label ::= identifier
il_simple_operation ::= ( il_simple_operator [il_operand] )
| ( function_name [il_operand_list] )
il_expression ::= il_expr_operator '(' [il_operand] EOL {EOL}
[simple_instr_list] ')'
il_jump_operation ::= il_jump_operator label
il_fb_call ::= il_call_operator fb_name ['('
(EOL {EOL} [ il_param_list ]) | [ il_operand_list ] ')']
il_formal_funct_call ::= function_name '(' EOL {EOL} [il_param_list] ')'
il_operand ::= constant | variable | enumerated_value
il_operand_list ::= il_operand {',' il_operand}
simple_instr_list ::= il_simple_instruction {il_simple_instruction}
il_simple_instruction ::=
(il_simple_operation | il_expression | il_formal_funct_call)
EOL {EOL}
il_param_list ::= {il_param_instruction} il_param_last_instruction
il_param_instruction ::= (il_param_assignment | il_param_out_assignment)
',' EOL {EOL}
il_param_last_instruction ::=
( il_param_assignment | il_param_out_assignment ) EOL {EOL}
il_param_assignment ::= il_assign_operator ( il_operand | ( '(' EOL {EOL}
simple_instr_list ')' ) )
il_param_out_assignment ::= il_assign_out_operator variable


il_simple_operator ::= 'LD' | 'LDN' | 'ST' | 'STN' | 'NOT' | 'S'
| 'R' | 'S1' | 'R1' | 'CLK' | 'CU' | 'CD' | 'PV'
| 'IN' | 'PT' | il_expr_operator
il_expr_operator ::= 'AND' | '&' | 'OR' | 'XOR' | 'ANDN' | '\&N' | 'ORN'
| 'XORN' | 'ADD' | 'SUB' | 'MUL' | 'DIV' | 'MOD' | 'GT' | 'GE' | 'EQ'
| 'LT' | 'LE' | 'NE'
il_assign_operator ::= variable_name':='
il_assign_out_operator ::= ['NOT'] variable_name'=>'
il_call_operator ::= 'CAL' | 'CALC' | 'CALCN'
il_return_operator ::= 'RET' | 'RETC' | 'RETCN'
il_jump_operator ::= 'JMP' | 'JMPC' | 'JMPCN'
```

3.3 Production rules for the elements of the ST programming language

In this section, the production rules that describe the elements of the ST programming language are presented, which correspond to *expressions* and *sentences*.

```
expression ::= xor_expression {'OR' xor_expression}
xor_expression ::= and_expression {'XOR' and_expression}
and_expression ::= comparison {('&' | 'AND') comparison}
comparison ::= equ_expression { ('=' | '<>') equ_expression}
equ_expression ::= add_expression {comparison_operator add_expression}
comparison_operator ::= '<' | '>' | '<=' | '>=' '
add_expression ::= term {add_operator term}
```

```
add_operator ::= '+' | '-'
term ::= power_expression {multiply_operator power_expression}
multiply_operator ::= '*' | '/' | 'MOD'
power_expression ::= unary_expression {'**' unary_expression}
unary_expression ::= [unary_operator] primary_expression
unary_operator ::= '-' | 'NOT'
primary_expression ::=
constant | enumerated_value | variable | '(' expression ')'
| function_name '(' param_assignment {',' param_assignment} ')'


statement_list ::= statement ';' {statement ';'}
statement ::= NIL | assignment_statement |subprogram_control_statement
| selection_statement | iteration_statement

assignment_statement ::= variable ':=' expression

subprogram_control_statement ::= fb_invocation | 'RETURN'
fb_invocation ::= fb_name '(' [param_assignment {',' param_assignment}]')'
param_assignment ::= ([variable_name ':='] expression)
| (['NOT'] variable_name '=>' variable)

selection_statement ::= if_statement | case_statement
if_statement ::=
'IF' expression 'THEN' statement_list
{'ELSIF' expression 'THEN' statement_list}
['ELSE' statement_list]
'END_IF'
case_statement ::=
'CASE' expression 'OF'
case_element
{case_element}
['ELSE' statement_list]
'END_CASE'
case_element ::= case_list ':' statement_list
case_list ::= case_list_element {',' case_list_element}
case_list_element ::= subrange | signed_integer | enumerated_value

iteration_statement ::=
for_statement | while_statement | repeat_statement | exit_statement
for_statement ::=
'FOR' control_variable ':=' for_list 'DO' statement_list 'END_FOR'
control_variable ::= identifier
for_list ::= expression 'TO' expression ['BY' expression]
while_statement ::= 'WHILE' expression 'DO' statement_list 'END_WHILE'
repeat_statement ::=
'REPEAT' statement_list 'UNTIL' expression 'END_REPEAT'
exit_statement ::= 'EXIT'
```

4 Conclusions

The syntactic specifications for programming languages of the IEC 611131-3 standard have been presented. This specification was described through of the production rules, which define the different elements of these programming languages. Then, the common production rules for textual and graphical languages and the specific rules for the textual programming languages (IL and ST) have been presented. Since the syntactic specification for graphicals programming languages (LD and FBD) is already developed, and the start symbol and the sets of terminal and nonterminal symbols are contained in these production rules, it is possible to define the CFG that generates the programming languages provided by IEC 611131-3 standard.

The importance of defining a CFG for the development of a translator that can generate code in a high-level programming language (C programming language) from a program written in any of the programming languages described by the IEC 61131-3 standard lies in the fact that it is possible the design and implement a lexical analyzer, since the regular expression that describes a programming language are a particular case of the CFG. Also, a syntactic analyzer can be designed and implemented from this grammar, which finally determines if a sentence belong or not to the programming languages generated by the the proposed CFG. For the implantation of the translator is proposed to use of open standards and open source, which represent an important contribution because currently most of the translators for programming languages for PLC, that are available in the market, are developed using licensed software, closed code or not free code. The proposed translator has all the kindness of open source.

As a future work, it is neccesary the description of the production rules that allow to make the required combinations to specify the SFC programming language of the IEC 611131-3 standard.

References:

[1] I. 61131-1. *Programmable controllers - Part 1: General information*. Norma Internacional, IEC (International Electrotechnical Comission), 2003.

[2] I. 61131-1. *Programmable controllers - Part 3: Programming languages*. Norma Internacional, IEC (International Electrotechnical Comission), 2003.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 2nd edition, 2007.

[4] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[5] F. Hidrobo, A. Ríos, G. Díaz, I. Besembel, F. Narciso, O. González, Y. Rivas, A. Del Mar, and B. Nava. Quinto Informe: Diseño de una Herramienta Computacional para el Desarrollo de las Lógicas de Control Basada en la Norma IEC 61131-3. Ingeniería de Detalle. Technical report, UAPIT-ULA, Mrida, Septiembre. 2009.

[6] F. Hidrobo, A. Ríos, G. Díaz, F. Narciso, I. Besembel, A. Del Mar, Y. Rivas, O. González, and B. Nava. Cuarto Informe: Diseño de una Herramienta Computacional para el Desarrollo de las Lógicas de Control Basada en la Norma IEC 61131-3. Technical report, UAPIT-ULA, Mrida, Mayo. 2009.

[7] F. Hidrobo, A. Ríos, F. Narciso, and I. Besembel. Diseñando un SoftPLC sin Uso de Licencias. In *V Congreso Colombiano de Computación , CCC-2010*, Abril 2010.

[8] H. Jack. Automating Manufacturing Systems with PLCs. Jack Books, 2005. http://claymore.engineer.gvsu.edu/jackh/books.html.

[9] D. M. Ritchie and B. W. Kernighan. *The C programming language*. Prentice Hall, Englewood Cliffs, NJ, 1988.

ISBN: 978-960-474-257-8