Edouard Tisserant, Laurent Bessard, and Mário de Sousa

An Open Source IEC 61131-3 Integrated Development Environment

Abstract—**The IEC 61131-3 standard defines a common framework for programming PLCs (Programmable Logic Controllers), which includes the complete definition of four programming languages and a state machine definition language. Industrial PLC vendors are slowly offering support for this standard, however small inconsistencies remain between their implementations, transferring programs between vendors is almost impossible due to different file formats, and licenses are generally too expensive to allow students do install these commercial solutions on their own computers.**

To this end, the authors have developed an Integrated Development Environment (IDE) for the IEC 61131-3 framework, which is being offered to the general public under the GNU Public License (GPL). The IDE consists of a Graphical User Interface (GUI) and a backend compiler. Using the GUI the user may develop programs in any of the four programming languages, as well as the state machine definition language. The backend compiler is used to convert these programs into equivalent C++ programs which may later be compiled and executed on various platforms.

I. Introduction

THE proliferation of PLCs (Programmable Logical Controllers) used in an industrial setting is indicative of their usefulness. These have evolved with the times, to the point that many modern top of the range PLCs are actually full fledged computers in disguise, executing modern operating systems. The hardware inside many vendor's PLCs has a tendency to become similar to PCs (Personal Computers) in order to take advantage of economies of scale.

On the other hand, the diversity of the programming languages used between different vendors, along with their increased complexity, has led to larger learning times for the programmer when switching between PLC. To this end, the IEC (International Electrotechnical Commission), an international standards body, has approved a collection of standards with the intention of creating a common user experience when configuring and programming industrial controllers. One of the components of this standard, namely the IEC 61131-3 [1], defines how the user may program the PLCs, and includes a programming framework and several programming languages.

Nevertheless, even though both the hardware and the software aspect of differing vendor PLCs are becoming similar, the vendors are still able to lock users into their line of products using several techniques which we shall not discuss.

Additionally, and just like any other programming language, learning and becoming proficient with these programming languages requires that the student practice extensively program development and therefore learn from his/her mistakes and errors. Expensive licensing deals from existing vendors makes it cost prohibitive to allow students to install an IEC 61131-3 programming environment on their own personal computers. Practice is therefore limited to the few workstations that may be made available to the students and located on the school campus.

The authors therefore embarked on the project of developing and IDE for the IEC 61131-3 standard that may be used freely and without restrictions by anybody who so wishes. In order to foment the dissemination and take advantage of any help that third parties may wish to provide, the code has been made publicly available under the GPL. The project has two main requirements, namely:

- strict adherence to the IEC 61131-3 standard
- cross platform support

A. Outline

This paper describes the implementation of an IEC 61131-3 integrated development environment. After this first section which introduces the paper, section 2 gives a necessarily brief overview of the IEC 61131-3 standard. Section 3 describes the graphical user interface of the IDE and its implementation, and section 4 describes the IL and ST compiler that comprises the backend. We conclude in section 5 with a few comments on the IEC 61131-3 standard itself, and point to directions to which we may draw our attention in future work.

II. IEC 61131-3 Overview

The IEC 61131 standard [1] is a general framework, that tries to establish the rules to which all PLCs should adhere to, encompassing mechanical, electrical, and logical aspects. The third part, IEC 61131-3, deals with the programming aspect of the industrial controllers, defining the logical programming blocks and the programming languages.

There are three variations of top level programming blocks: functions, function block types, and program types. Functions have similar semantics to those in traditional functional languages, and directly return a single output value. However, besides one or more input values (equivalent to variables passed as values), the function may also have parameters used as outputs (equivalent to passing variables as references), or as input and output simultaneously. Function semantics state that they are idempotent, i.e. all invocations of the same function with

Manuscript received January 24, 2007.

Mário de Sousa is with the Electrical Engineering Department, University of Porto, 4200 465 Porto, Portugal, (phone: +351 22 508 1815; fax: +351 22 508 1443; e-mail: msousa@fe.up.pt).

Edouard Tisserant and Laurent Bessard are with TBI SARL - Lolitech, 88100 Saint-Dié-des-Vosges, France (phone: +33 (0)3 29 52 95 67; e-mail: edouard.tisserant@lolitech.fr, laurent.bessard@lolitech.fr).

the same input values should always yield exactly the same result, whatever the state of the rest of the system, including the time at which the function is executed.

Function block types are similar to classes in object oriented languages, with the limitation of having a single public member function. Function blocks are instantiated as variables, each with their own copy of the function block state. The default function of a function block does not directly return any value, but, like functions, may have parameters to pass data as input, output or bidirectional. Since the function block has only a single function, calling this function is commonly referred to as 'calling the function block'. Likewise, this function's parameters are often referred to as the function block parameters.

Since a function must be idem-potent, it can neither instantiate nor call a function block instance. It may, however, read the current values of the output or bidirectional parameters of a function block. Note that the function block instance must be passed to the function as an input parameter, as a function may not instantiate a function block instance.

Program types are very similar to function blocks, with the exception that these may only be instantiated inside a configuration, and not inside other functions, function block types or program types.

A configuration is the program organization unit with the highest abstraction level. It does not contain executable code, but rather instantiates programs and/or function blocks, creates and configures tasks, and assigns the programs and/or function blocks to tasks. Tasks are similar to processes in common operating systems, and may have periodic execution or execute upon the occurrence of the rising edge of a specified Boolean variable.

The three types of programming blocks may be programmed in one of two textual languages (IL - Instruction Language; ST - Structured Text), or two graphical languages (LD - Ladder; FBD - Function Block Diagram).

The standard also defines a graphical language for specifying state machines (SFC – Sequential Function Chart), mostly based on Grafcet, that may also be used in programming function blocks or programs. Since a state machine implies the maintenance of state, SFCs may not be used to program functions because these must be idem-potent. It should be noted that without reverting to the other languages it is not possible to write a complete program using only an SFC chart, which is why the authors hesitate to refer to SFC as a programming language.

III. The Graphical User Interface

The user interacts with the IEC integrated development environment through a graphical interface. This graphical interface lets the user create a project consisting of several IEC 61131-3 program organization blocks (POUs). These POUs are listed in a tree view on the left pane of the IDE. Each POU in the tree may be expanded to show its

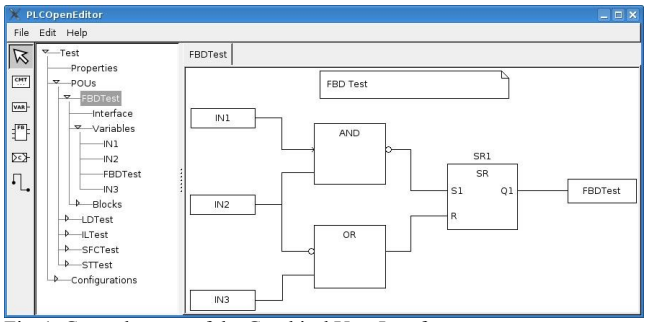

Fig. 1. General aspect of the Graphical User Interface

interface variables, as well as all internal variables (*Fig. 1*).

Each POU may be further programed in any of the IEC 61131-3 languages, using the appropriate language editor on the right pane of the IDE. The definition of the POU interface, as well as all internal variables is made through a graphical window interface (*Fig. 2*).

A. The PLCOpen editor

The GUI lets users program with the five languages defined by IEC 61131-3 standard :

- Sequential Function Chart (SFC)
- Function Block Diagram (FBD)
- Ladder Diagram (LD)
- Structured Text (ST, equivalent to C/C++)
- Instruction List (IL, equivalent to assembler)

The graphical editor is strongly linked to PLCOpen specification [2]. This specification defines an XML grammar describing the five IEC 61131-3 languages. All automation programs written in this environment are saved into XML files, according to this grammar. It is then possible to exchange projects with other IEC 61131-3 editors that conform with the PLCOpen standard.

The graphical editor is written with python, and uses the python binding to wxWidgets. The use of these two technologies allows the code to be portable between different platforms, including Windows and Linux.

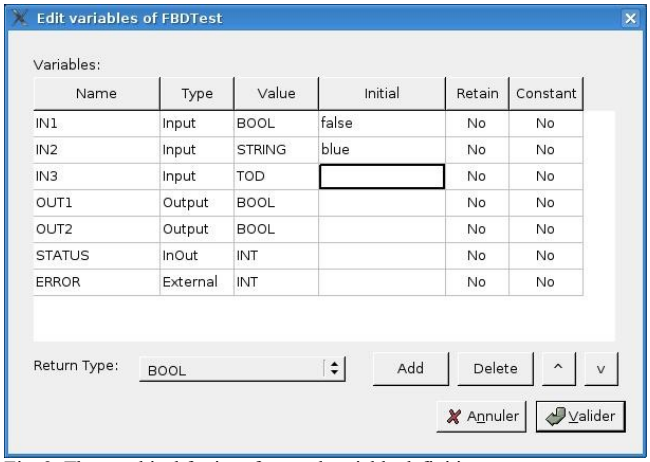

Fig. 2. The graphical for interface and variable definition

All program editors follow the MVC (Model-View-Controller) paradigm. The object classes used by the first

component *(i.e.* the Model*)* are dynamically generated from the official scheme (.xsd) defined in the PLCOpen specification. Incorporating future changes of the PLCOpen specifications into the model component of the editor may therefore be automated.

The SFC, FBD and LD graphical editors allow the user to insert and delete programming elements in such a way as not to permit the user to introduce illegal layout. These programs are therefore always in a correct, although possibly incomplete, state.

The SFC editor (*Fig. 3*) provides a toolbox with which to insert initial steps, steps, transitions, and transition convergences or divergences. The insertion of these elements (except an initial step) must always be referenced to a previously existing element of the SFC. For example, to insert a transition the user must first select the step to which it will be associated. Likewise, to insert a step the user must first select a transition or another step. In the case a step is selected, the editor automatically inserts a transition between the steps.

The LD editor (*Fig. 4*) follows the same philosophy. The creation of a new rung implies the insertion of an output relay. The rung is always in a consistent state, only allowing the user to introduce new elements in such a way as to produce another valid state.

The textual language editors, IL (*Fig. 5*) and ST (*Fig. 6*), include syntax highlighting of the code, and autocompletion of keywords and variable names. Simple syntax errors are highlighted, nevertheless (and unlike the graphical editors), the code may be saved with the syntax and/or semantic errors inside.

B.Conversion of Graphical Languages

A module is responsible to translate PLCOpen graphical language (FBD and LD) into ST. This part is integrated into the graphical editor, but may be used independently. The reverse value propagation algorithm is used to convert these graphical languages into ST.

Conversion is also conditioned by optional debug mode, that adds necessary information in generated code. This information will then be used at runtime to ensure status feedback for users.

Graphical SFC programs, on the other hand, may be converted to the textual syntax used to express SFC programs. This textual syntax, although not commonly used, has been normalized in IEC 61131-3. As with the textual ST and IL languages, it will be up to the backend compiler to compile SFC programs expressed using the textual syntax into the equivalent C++ program.

C.Human Machine Interface Creation Tool

The authors intend to continue the development of the project in several fronts. One these will be the integration of a tool (which the authors intend to call SVGUI) to allow the user/programmer to define graphical interfaces to the automation control program. This will be based on the "SVG" (Scalable Vector Graphics) open W3C standard,

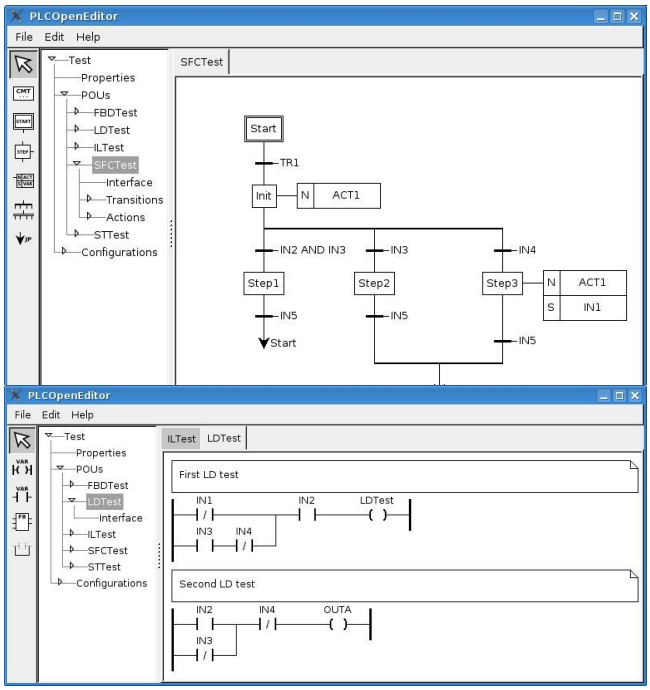

Fig. 4. The LD editor

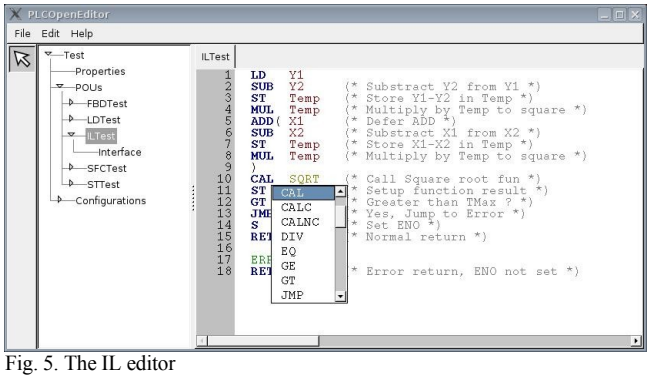

Fig. 5. The IL editor

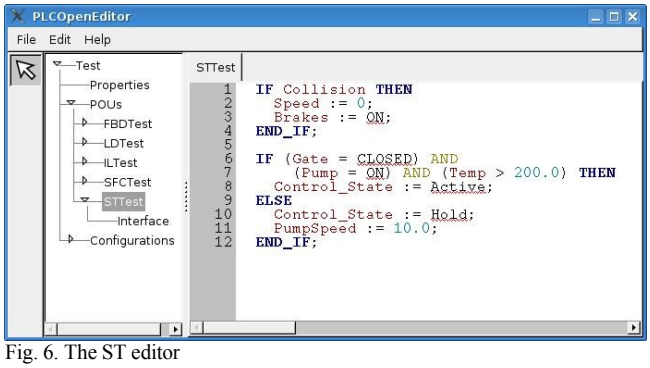

Fig. 6. The ST editor

and XML grammar for describing vector graphics. The main idea is to let users "draw" their HMI with any standard SVG drawing software such as Inkscape, and select graphical elements that will participate in the interaction with the automation program.

The interaction will be made through the use of wxSVG, which is a SVG renderer using wxWidgets graphical library. The particularity of wxSVG is that it loads an SVG

file into a "living" object tree in memory, allowing "live" access to the graphical elements.

A "widgets" library (e.g: buttons, scrollbars, textareas, checkboxes,...) will have to be developed that uses wxSVG graphical elements for representation and interaction. As an example a rectangle drawn under Inkscape can be defined as a button and when clicked, change form and colour, and move around the screen.

In SVGUI, appearance is completely independent of code. It is possible to change GUI appearance without any source code change. This permits the graphical creation to be entrusted to a graphical designer (i.e. non-programmer) and the interaction to a programmer. This aspect is actually very difficult to find on the majority of GUI toolkits.

The link between SVG graphical elements (rectangles, circles, ...) and widgets (buttons, textarea, ...) is done through a simple XML file, that declares which graphical elements participate with which widget. Widgets have their own names and specific variables. All the widgets used will automatically generate a corresponding Function Block that may be used by the automation program. Each function block instance will correspond to a single widget instance, and each pin to a variable.

Using this architecture, SVGUI will be a library written in C++. Development of some "bindings" for other languages should be rather straightforward.

IV. The Backend Compiler

The backend portion of the IDE consists of a compiler that converts IL, ST and SFC programs into equivalent C++ programs. This compiler executes in four plus one stages: lexical analyser, syntax parser, semantics analyser, code generator, and binary 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 data structure. The semantic analyser walks through the abstract syntax and determines whether all semantic rules have been obeyed. The code generator, based on the abstract syntax once again, produces the final equivalent 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. At the moment we have merely implemented a C++ code generator. The last stage of the overall compiler will generate the final executable from the code generated from the previous stage. In our case we are currently using the gcc compiler to generate the final executable.

Our abstract syntax tree has been implemented as a tree of objects that follow the visitor design pattern [3]. This enables us to easily add or remove stages to our architecture without having to edit the abstract syntax tree classes themselves. Possible additions to the architecture include a code optimization stage.

A. 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 nevertheless still has its difficulties. The main issue is the definition of the EOL token (used in the IL language), and defined by the standard as “normally consisting of the 'paragraph separator' character defined as hexadecimal code 2029 by ISO/IEC 10646 “. This statement seems to leave to the implementors the final choice of which character should represent the EOL token. It is our intention to allow the programs to be written using existing text editors that generate ASCII text. However, the suggested character is not an ASCII character, and we were therefore forced to choose another character to represent the EOL token. The 'newline' (ASCII 10hex) character seemed to be the most natural.

This choice, although seemingly obvious, complicates matters since the newline character is considered whitespace in the standard. This means that the lexical analyser may only generate EOL tokens while parsing IL statements, and ignore it otherwise. Our solution was to implement a state machine in the lexical analyser, with a very limited knowledge of the syntax, that tracks whether IL statements are currently being parsed, and therefore parsers the newline character as the EOL token, instead of the normal whitespace.

However, the state machine got more complex when we decided to allow our compiler to automatically detect whether it was parsing IL, ST or SFC, the reason for which is explained later.

B. 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, of which we shall mention only a few.

As the IL and ST languages share a very large common syntax related to the declaration of types, functions, variables, etc., we decided to write a single parser that would handle both languages simultaneously. It was this choice that led to the more complex state machine in the lexical parser, which has already been mentioned.

A few conflicts were found in the syntax definition given in the standard. These were mostly due to the existence of more than one route for reducing several constructs. All of these conflicts were easily resolved, as the expected semantics of either route were identical.

Other more thorny issues were related to the fact that the language requires more than one look ahead token to be correctly parsed, whereas the bison utility generates a parser that uses a single look ahead token. We worked

around this issue by reducing to temporary constructs until the look ahead token that would break the deadlock was available. Only then was the temporary construct changed to the correct final construct. This of course resulted in a much more confusing configuration file for the bison utility, and less maintainable code.

We also found that the syntax does not contain sufficient redundancy to allow it to be successfully parsed without the help of a symbol table that keeps track of the type of construct to which an identifier (name) refers to. The syntax parser therefore makes use of two symbol tables, one for the global references (function names, data type names, ...), and another for variables declared within functions, programs or function blocks. The syntax parser adds entries to these tables when an identifier is declared. Subsequently, when the lexical parser comes across an identifier, it will first look it up in these tables to verify if it has been previously declared. Before any parsing commences, the global table is initialised with the names of all the default functions and functions blocks defined in the standard.

The use of the symbol tables results in many semantic checks being inherently performed by the syntax parser. For e.g., a variable name may only be used if it has been previously declared. Type checking, for e.g., is nevertheless not performed, and is left to the semantic checker.

C.The Semantic Checker

Due to a lack of time and resources, the semantic checker has not yet been implemented. Although extremely important for the correct functioning of the overall compiler, we felt that this part could be left for a later stage. This is mainly due to the fact that our first code generator creates C++ source code, which is then compiled by the gcc compiler. Many semantic errors in the ST or IL source code will also result in semantic errors in the C++ source code, which will be caught by the gcc compiler. Some semantic errors, such as calling a function block from within a function, will nevertheless not get caught.

This is of course not a desirable scenario for a fully functioning compiler, as the user does not get any feedback as to the location of the error in the ST or IL source code. It is therefore our intention to complete the semantic checker as soon as possible.

D.The C++ Code Generator

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++. Data types not supported directly by C++, such as time of day, were implemented as specific C++ classes, with overloaded operators.

Contrary to what might be expected, the FB and program type constructs are both mapped onto a C++ structure data type that contains all internal and interface variables for the FB or program, and an accompanying function that takes an instance of the referred data structure as its single parameter. This was done in order to allow the loading (LD operator of the IL language) of FB instances, which requires that the FB instance be copied onto a default accumulator type variable, that must be able to store many different data types. This accumulator variable is therefore implemented as a C++ union.

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 a configuration is mapped to C++.

Due to time constraints, and in order to maintain program portability, complete mapping of configuration constructs has not yet been implemented. Currently a single task running a single program instance is supported. This single program may however call as many FB or functions that may be necessary.

V.Conclusions

We believe that the editor and compiler are already usable in both an academic and industrial environment. We expect to put it to the test in the next academic year.

Nevertheless, some work still remains to be done. Future work will involve writing the semantic checker, and defining a runtime to allow the full syntax of configurations, for at least one platform. Work on the SVGUI platform to support graphical user interfaces has already started.

References

[1] IEC, “IEC 61131-3, 2nd Ed. Programmable Controllers – Programming Languages”, International Electrotechnical Comission, 2003

[2] PLCopen Technical Committee 6, “XML Formats for IEC 61131-3, Ver 1.0”, April 2005

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design Patterns”, Addison-Wesley, 1997, IEBN 0-201-63361-2