graphics editors in CPDev environment

Marcin Jamro
Rzeszow University of Technology, Department of Computer and Control Engineering, Poland
mjamro@kia.prz.edu.pl

Abstract: According to IEC 61131-3 norm, controllers and distributed control systems can be programmed in textual and graphical languages. In many scenarios using a graphical language is preferred by the user, because diagrams can be more legible and easier to understand or modify also by people who do not have strong programming skills. What is more, they can be attached to the documentation to present a part of a system implementation. CPDev is an engineering environment that makes possible to program PLCs, PACs, softPLCs and distributed control systems with the usage of languages defined in IEC 61131-3 norm. In earlier versions, it supported only textual languages – ST and IL. Currently, graphics editors for FBD, LD and SFC languages are also available, so users can choose a suitable language depending on their skills and a specificity of a program that they have to prepare. The article presents implementation of the graphics editors, made by the author, which support creating program organization units in all graphical languages defined in IEC 61131-3 norm. They are equipped with a set of basic and complex functionalities to provide an easy and intuitive way of creating programs, function blocks and functions with visual programming. In the article the project structure and some important mechanisms are described. They include e.g. automatic connections finding (with A* algorithm), translation to ST code, conversion to and from XML format and an execution mode supporting multiple data sources and breakpoints.

Keywords: IEC 61131-3, graphical languages, visual programming, control systems

1. Introduction

CPDev (Control Program Developer) [12] is an engineering environment which can be used for programming PLCs (Programmable Logic Controllers), PACs (Programmable Automation Controllers), softPLCs (PCs used as controllers) and distributed control systems according to IEC 61131-3 norm [10]. It has been being developed in the Department of Computer and Control Engineering at Rzeszow University of Technology for a few years. CPDev is used not only by lecturers and students during didactic activities but also by companies, like LUMEL S.A. (Poland) [8][9] and Praxis Automation Technology (Netherlands) [11] which uses it in ship control and monitoring system. Cooperation with industry has an important impact on improving CPDev, because it shows required new features and areas where the software can be improved. After creation of the first version, some possible disadvantages were found, that should be removed in the next version. One of necessary features is a support for developing programs in FBD, LD and SFC languages [7], which was designed, implemented and connected with CPDev environment by the author.

In the current version users can create program organization units (POUs) in all languages defined in IEC 61131-3 norm, i.e. ST (Structured Text), IL (Instruction List), FBD (Function Block Diagram), LD (Ladder Diagram) and SFC (Sequential Function Chart). It
makes possible to choose a programming language by the user depending on skills and problem specificity.

Programming in graphical languages has many advantages. One of them is diagrams legibility which makes understanding of an implementation easier and faster. Users can also modify it in an easier way. Usage of graphical languages could require less work and be more comfortable for programmers than using textual languages, especially for those who do not have strong programming skills. What is more, diagram printouts can be used as a part of the project documentation.

Some integrated development environments with editors of graphical languages already exists, e.g. CoDeSys [4], Beckhoff TwinCAT [3], Control Builder F [1] or even an open-source environment [14]. However, these solutions can’t be easily integrated with CPDev software and its modules, like CPSim. The solution which met this requirement is necessary for the further development of CPDev environment. It caused an implementation of new graphics editors equipped with many advanced possibilities, not always available in other IEC 61131-3 IDEs, like an automatic connections finding or an execution mode with support for multiple data sources, tracing variable values or even breakpoints (both conditional and unconditional). Designing and implementation of graphics editors of FBD, LD and SFC languages, dedicated to CPDev environment, made possible to get many benefits caused by strict integration with the existing software and its modules.

The second chapter presents a structure of graphics editors projects with information about common features, classes and interfaces. Main implemented mechanisms available in all graphics editors are explained in the third chapter.

2. Graphics editors projects

Adding support for graphical languages [6] caused some modifications in a concept of the whole CPDev environment (Figure 1). FBD, LD and SFC diagrams are translated to the ST code, which is compiled to VMASM. It is assembled to the virtual machine code, that is run on target platforms containing CPDev virtual machine. An addition of an intermediate stage (translation to the textual ST language) is caused by an availability of well-tested ST compiler in CPDev environment. By this design, programs created in graphics editors can be run on all platforms supported by CPDev. It significantly increases software functionalities and eliminates a chance of incorrect work of FBD, LD and SFC compilers.

![Figure 1. A concept of CPDev environment](image)

Apart from translation to ST code, it is also necessary to prepare a mechanism to save an exact state of the diagram as a string, which can be used to load diagram without losing any data. To achieve this goal, XML format is used. Its structure is based on PLCopen standard [13] with modifications required by additional features implemented in the graphics editors.
Mechanisms of translation and conversion cooperate to create a complete solution for saving diagram, which can be then translated to ST code, compiled and run on any target platform supported by CPDev (Figure 2).

![Diagram conversion XML translation ST](image)

Figure 2. A cooperation between mechanisms of diagram conversion and translation

FBD, LD and SFC editors are prepared in C# language with the usage of .NET Framework 2.0 (to provide developers with consistency with the main part of CPDev software) and Microsoft Visual Studio 2010 integrated development environment. Every editor is written as a separate project (FBDEditor, LDEditor, and SFCEditor) connected with a general library called GraphicEditor. It contains implementation common for all graphics editors including a mechanism of automatic connections finding, a class of a main editor’s window or classes and interfaces representing diagram elements.

### 2.1. Features

All editors of graphical languages in CPDev are equipped with a set of common functionalities necessary for creation of program organization units (POUs) with the usage of elements available in a given programming language. An important assumption is to prepare graphics editors which allow users to create diagrams as fast as possible and in an intuitive way. It causes also a necessity of adding solutions dedicated for specific editors. A set of basic functionalities contains:

- adding, removing and moving elements,
- copying, pasting and cutting elements,
- saving and loading diagrams,
- automatic connections finding between outputs and inputs with an automatic update after changing a position or a size of an element,
- translation of the diagram to ST code,
- conversion of the diagram to and from XML format,
- printing the diagram accordingly to the printout template,
- setting element properties,
- automatic adjustment of elements width,
- adding branch points,
- detection of lines intersection,
- operations history (undo and redo commands),
- adjusting display settings for diagram, e.g. by showing or hiding grid and subsidiary lines or changing scale,
- simple diagram verification,
- informing a user of an attempt of placing an element in an incorrect location,
- an execution mode for running programs with a support for tracing variable values and breakpoints (conditional and unconditional).

Particular editors contain also a set of dedicated features, that make creating solutions with them easier and faster, like:

- drawing lines on FBD diagrams in different colors depending on their types and settings (line color, width and style) defined by the user,
- automatic generation of rungs on LD diagrams,
Marcin Jamro

- a possibility of adding elements directly on existing rungs on LD diagrams,
- creating actions in SFC language with the usage of ST, IL, FBD and LD editors,
- automatic generation of vertical lines on SFC diagrams.

2.2. Common classes and interfaces

Many elements and mechanisms are similar in editors of different graphical languages available in CPDev environment. They are extracted as common classes and interfaces in an additional GraphicEditor library which is referenced by projects representing FBD, LD and SFC editors.

All of them have a window used to present a single diagram (Figure 3). It is represented by an instance of a class deriving from FormDiagram which derive from CPDChild that enables cooperation between the editor and the main CPDev window [6]. The window with a single element has two main parts:

- left – containing a tree with nodes representing all elements which can be placed on the board (like variables, comments, functions or instances of function blocks), and a textbox to filter elements accordingly to a substring typed by a user,
- central – with a board.

Both parts are developed as controls named ItemsTree and DrawingPanel. Window appearance can be adjusted, which makes possible to add elements required by a given language.

![Figure 3. Main CPDev window](image)

In the window a diagram is shown, which is represented by an instance of a class deriving from GraphicDiagram. It contains a few properties to set and get diagram data, including a list of drawn elements (Elements), a list of all types available in the project (Types), content type (program, function or function block – ContentType), and data of global variables existing in the project (GlobalVariables). It contains also methods making possible to perform operations on the diagram (created in any graphical language) like adding or removing an element (AddElement and DeleteElement) and getting selected elements (GetSelectedElements).

An important group of classes and interfaces is related to elements which can be placed on the diagram, including functions, instances of function blocks, and lines (Figure 4). These elements can be grouped by some abilities which is well visible in implementation of particular interfaces and makes possible to perform operations on different elements in the same way if they have the same ability.
All elements which can be placed on the diagram implement `IDrawable` interface (Listing 1). It defines basic properties and methods, including rectangle which represents an element (`Rectangle`), value indicating whether an element is currently selected (`IsSelected`) and a method to draw it on the board (`Draw`).

```csharp
public interface IDrawable
{
    Rectangle Rectangle { get; set; }
    Rectangle CollisionRectangle { get; }
    bool IsSelected { get; set; }
    bool IsInvalid { get; set; }
    ElementContextMenu Menu { get; }
    void Draw(Graphics graphics, double scale, DiagramModeEnum mode);
    void Deselect();
    List<IDrawable> GetAllDrawableElements();
    IDrawable GetElement(Point point);
    IDrawable CloneElement(Clones clones);
}
```

Listing 1. `IDrawable` interface

`IParent` interface is implemented by elements which contain inputs or outputs and are distinguished by an identifier (`LocalId`, an integer value). These elements contain functions, instances of function blocks and branch points. Classes that implement `IParent` have to possess members of both `IDrawable` and `IParent` interfaces. The last one defines methods to adjust dynamically a size of element (`PrepareRectangle`) and to set a required margin around it (`GetMapModifiers`).

Another group consists of elements which are start or end parts of a line that form a connection. Classes representing elements from this group implement `IConnectable` interface which defines properties pointing to a previous and next element (`PreviousElement` and `NextElement`) and also a method returning a location of a starting point for the next line (`GetConnectionPoint`). A connection does not always contain only one line, because if direction changes are required, the connection will contain more lines (Figure 5). Classes implementing `IConnectable` interface represent for instance inputs and outputs of elements placed on the diagram (`Input` and `Output`) and lines (`Line`).

An important element existing on diagrams created in FBD and LD languages is a branch point. It is represented by an instance of `BranchPoint` class implementing `IConnectable` interface. This element makes possible to split or connect lines, which is necessary in many scenarios, for instance to perform OR operation on LD diagrams (Figure 7). An additional class related to branch points is `BranchPointInputOrOutput` representing its input or
output. Four its instances are connected with every instance of BranchPoint class. They represent places where connections can be added, and are located above, below, on the left and on the right of the branch point (Figure 6).

![Figure 5. Differences between line and connection](image)

![Figure 6. Graphical representation of a branch point.](image)

![Figure 7. An example of LD program which contains OR operation](image)

3. Mechanisms available in graphics editors

Graphics editors available in CPDev environment are equipped with many mechanisms to perform different operations related to either diagram or the window in which it is presented. These mechanisms allow to:

- find connections between elements placed on the diagram,
- convert diagram to and from XML format,
- translate diagram to ST code,
- verify diagram,
- create a copy of elements,
- create instances representing different diagram elements,
- support operations history, including undo and repo operations,
- execute a program, either in simulation or commissioning,
- perform operations on elements map,
- show useful information to the user,
- show tooltips for elements placed on the diagram,
- print diagram accordingly to the printout template,
- analyze frequently used elements.

Instances representing all mechanisms mentioned above are values of properties of FormDiagram class. This solution hides details of implementation, improves code readiness and makes possible to modify it in an easier way.

3.1. Automatic connections finding

One of the assumptions of graphics editors is to make possible to create program organization units without unnecessary work. It is achieved also by implementation of the mecha-
nism of automatic connections finding, for instance between an output of function block instance and a variable input. It is important to be certain that a connection:

- can be found every time (if elements are placed on the diagram correctly),
- passes round elements added earlier,
- rarely changes directions,
- limits a number of intersections with other connections (Figure 8).

Figure 8. Examples of connections
– ToDiagram – converting the XML format to the object representing the diagram. Converter class has also an implementation of methods used to convert parts common for all languages, like WriteFileHeader, WritePositionTag, and ReadComments. Implementation details for specific languages are set in classes deriving from Converter, placed in projects of different editors.

```xml
(a) <inVariable height="20" 
localId="0" width="60">
  <position x="40" y="180" />
  <connectionPointOut>
    <relPosition x="70" y="-10" />
  </connectionPointOut>
  <expression>
    ALARM/expressions
  </inVariable>

(b) <step localId="0" height="40" 
width="235" name="OFF"
initialStep="true">
  <position x="90" y="30" />
  <connectionPointIn>
    <relPosition x="70" y="-10" />
    <connection formalParameter=""
      refLocalId="" />
  </connectionPointIn>
  <connectionPointOut>
    <relPosition x="70" y="50" />
  </connectionPointOut>
</step>
```

Listing 2. Generated nodes of XML document: (a) inVariable (FBD), (b) step (SFC)

3.3. Translation to ST code

According to the assumption described in the second chapter, all diagrams are translated into ST code which is then compiled into VMASM code with the usage of ST compiler available in CPDev environment (Figure 1). This approach requires a mechanism of translating all diagrams to ST code in the same way. An additional interface named ITranslator is created to solve this problem. It contains only Translate method with one parameter of string type (diagram data in XML format) that returns a code in ST language prepared in a translation process.

Translation process depends on a language. From this reason, every project has a class implementing ITranslator interface with a logic of diagram translation to ST code.

![Figure 10. Translation of the FBD diagram to ST code](image-url)
In case of FBD program, information about global variables is placed in VAR_EXTERNAL part (Figure 10, area 1). Local variables including instances of function blocks and variables representing element outputs (with added connections) are located in VAR part (area 2 and 3). The main part of ST code contains instructions of calling instances of function blocks, functions and also setting values of output variables (area 4).

3.4. Drawing diagrams

All graphics editors need to draw elements on the board. It is implemented with the usage of methods available in Windows Forms technology [15], that makes possible to create user interfaces in a fast and easy way. It contains many classes representing window elements like buttons, textboxes and drop down lists.

When the board is being refreshed, a new bitmap and Graphics object are created. Then the mechanism prepares a rectangle representing a currently visible part of the diagram accordingly to the current scrollbars positions. The next step consists of drawing all necessary elements on the diagram and is performed by DrawDiagram method. At the end, bitmap is drawn on the board and presented to the user.

*DrawDiagram* method is defined as a virtual with a common implementation in FormDiagram class (from GraphicEditor library). It calls other methods that perform operations related to drawing the following diagram parts: grid, subsidiary lines, elements, connections, border around currently added element, temporary connections during moving elements, rectangle representing a selection, incorrect elements, additional diagram map, and symbols representing breakpoints. In case of specific languages, it is necessary to add some dedicated elements on the board, like left and right power rails on LD diagrams.

What is more, for all graphics editors a mechanism of double buffering is used to prevent from showing undesired effects while the board is being refreshed.

3.5. Printing diagrams

One of important advantages of graphical languages is diagrams legibility and a possibility of attaching their printouts directly to the project documentation to provide engineers with important information about implementation of a specific system part. FBD, LD and SFC editors are equipped with a mechanism of printing accordingly to the printout template defined by the user in XML file. It makes possible to adjust page margins and add a table with information about company name, project or printout date (Listing 3).

```xml
<?xml version="1.0" encoding="utf-8" ?>
<PrintingTemplate>
  <Borders>
    <Border Orientation="Landscape" Visible="True" Left="10" Top="10" Right="50" Bottom="40" />
    <Border Orientation="Portrait" Visible="True" Left="10" Top="10" Right="40" Bottom="50" />
  </Borders>
  <Table>
    <Row>
      <Label><Resource Name="Program" /></Label>
      <Content><Variable Name="PROGRAM" /></Content>
    </Row> (...)
  </Table>
</PrintingTemplate>
```

Listing 3. A part of printout template file
The printout template allows users to define content shown in the table in three ways: from resources, from variables (where the following keywords are available: \textit{PROGRAM}, \textit{PROJECT}, \textit{VERSION}, \textit{COMPANY} and \textit{CURRENT\_TIME}) or defined directly in the printout template file (with a support for many languages).

\section*{3.6. Simple diagram verification}

FBD, LD and SFC editors are equipped with a mechanism of finding basic errors on the diagram. The main concept is to connect a set of tests for each language and run them every time before a project build and on user demand (Figure 11). Tests analyze different diagram parts and save information in a form of errors and warnings. If errors exist, project can’t be built and messages are shown to the user with more information about error source.

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{diagram_verification.png}
\caption{Concept of the verification mechanism}
\end{figure}

An additional interface \textit{IValidator} is implemented by classes from \textit{FBDEditor}, \textit{LDEditor} and \textit{SFCEditor} projects. The interface has two lists with errors and warnings data represented by instances of \textit{ValidationMessage} class. By the usage of an additional class it is possible not only to show an information that error or warning occurred during diagram verification, but also point at elements that caused it. \textit{IValidator} interface (Listing 4) contains \textit{Validate} method which run all required tests. It is important that the mechanism of simple diagram verification cannot find all errors, but it could be useful for users to inform about operations which could be undesired and cause problems while the program is running.

\begin{lstlisting}[language=C#]
public interface IValidator
{
    List<ValidationMessage> Errors { get; }  
    List<ValidationMessage> Warnings { get; }  
    bool IsValid { get; }  
    void Validate(GraphicDiagram diagram);
}
\end{lstlisting}

\textit{Listing 4. IValidator} interface

\section*{3.7. Programs execution and testing}

All graphics editors contain an execution mode which allows users to check execution of created program either on a simulator or a real device (commissioning) by tracing variable values. The mechanism supports many data sources (Figure 12) that are systems providing current variable values, which can be retrieved for instance from simulator (local virtual machine), by Modbus protocol (used by Lumel S.A. company) or from FPGA platform.
Presentation of the received data on the diagram depends on its type and value (Figure 13). For instance, for variable of BOOL type and value FALSE a dashed line is drawn. In case of TRUE value, a line is solid. It is also possible to show values of variables as tooltips or a text above lines representing connections.

Execution mechanism supports breakpoints, both unconditional and conditional. They can be placed next to diagram elements (e.g. function block instances) and stop execution of the program just before performing an operation related to the element (e.g. setting a value of a variable or calling a function). In case of conditional breakpoints, the execution of the program will be stopped only if expression (given as a condition) is equal to TRUE.

4. Conclusion

IEC 61131-3 norm defines languages that make possible to program controllers and distributed control systems in a convenient way by using both textual and graphical languages. One of the most significant advantages is that users can choose a suitable language and combine different languages in the same project, e.g. write a function block in FBD and the main program in ST.

One of the integrated development environments used to program in languages of IEC 61131-3 norm is CPDev, which has been being developed in the Department of Computer and Control Engineering at Rzeszow University of Technology for a few years. Earlier it supported programming only in textual languages. That lack of functionality was removed after adding support, by the author, for graphical languages defined in IEC 61131-3 norm. Currently CPDev engineering environment can be used to create program organization units in any language of this norm. It significantly increases possibilities of its usage, because users can choose a suitable language depending on their skills and problem specificity.
There are many similarities between FBD, LD and SFC editors available in CPDev, which led to extraction of a common part as a separate GraphicEditor library containing a set of classes and interfaces used by other projects. It decreases an amount of code, number of potential errors and also makes modifications easier. Graphics editors are equipped with a set of mechanisms to perform specific operations on the diagrams, like automatic connections finding (with the usage of A* algorithm), translation to ST code, conversion to and from XML format, operations history, simple diagrams verification and also an advanced execution mode with support for tracing variable values and breakpoints.

References