AUTOMATYKA/ AUTOMATICS • 2013 • Vol. 17 • No. 1

http://dx.doi.org/10.7494/automat.2013.17.1.43

Marcin Jamro*, Bartosz Trybus*

Running and Testing the Programs Created in IEC 61131-3 Languages

1. Introduction

Software created in languages of IEC 61131-3 (PN/EN 61131-3) standard plays an important role in control and monitoring systems. For this reason, it is important to have an ability to test the software, find and remove faults before execution in real-time scenarios.

CPDev (*Control Program Developer*) (Stec *et al.* 2007) is an engineering environment used to create software for PLCs (*Programmable Logic Controllers*), PACs (*Programmable Automation Controllers*), softPLCs (PCs used as PLCs), and distributed control systems. CPDev is composed of a few modules, including editors for graphical and textual languages, project management module, and hardware configuration editor. The system is based on the virtual machine running on the target controller that makes created programs independent from specific hardware (Trybus 2011). In this paper, CPDev features for testing and executing of POUs (Program Organization Units) are described. Testing can be done either during simulation or in *on-line* mode. One of the newest mechanisms added to CPDev environment is the possibility of defining POU-oriented tests.

This paper is organized in the following way. Section 2 presents the data sources mechanism and the universal interface providing system with variable values for simulation and tracing. It is used by the program execution mode in graphics editors (described in section 3) and CPSim variables monitor (presented in section 4). The CPTest application for the creation and execution of unit tests and tests based on tables, is described in section 5.

2. Data sources mechanism

The CPDev environment makes it possible to trace the execution of programs running on a PC simulator (simulation mode) and on real devices (on-line mode, *commissioning*) (Jamro *et al.* 2011c). These possibilities are available in multiple CPDev modules, including editors of graphical languages (p. 3) and a CPSim application for tracing variable

* Rzeszow University of Technology, Department of Computer and Control Engineering, Rzeszow, Poland

values (p. 4). While running, they use the values of variables (data) provided either by the simulator or an external device (target controller).

The centralized data sources mechanism for variable monitoring has been implemented in CPDev (Fig. 1). It implements a universal interface for providing the project with current values of program variables. The simulator is one of the most important data sources for initial testing, because it uses a local CPDev virtual machine run on PC instead of the controller. Thus, it is possible to check program execution without the necessity of configuration of communication with the target device. Apart from the simulator, a data source with Modbus communication is available. It is used with a SMC industrial controller from LUMEL S.A. company. Another data source for CPCore FPGA multiprocessor controller is supported, with Intel-Hex communication protocol (Hajduk *et al.* 2011). A data sources mechanism can be easily extended due to the universal interface, which allows system designers to add their specific mechanisms providing current variable values. For example, it has been used by Praxis Automation Technology (Leiderdorp, Netherlands) that created a module for the *Mini-guard* ship control and monitoring system.

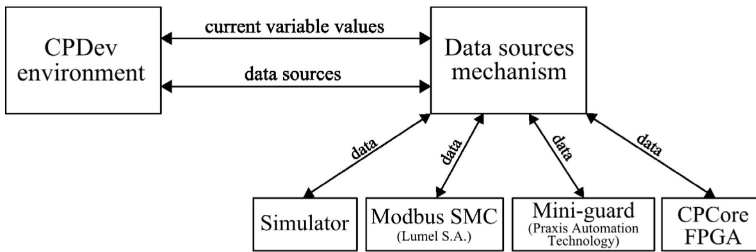

Fig. 1. Data sources mechanism

Apart from variable values, the data sources mechanism provides the CPDev environment with a list of available data sources. The user can then choose data sources or change their configuration parameters (such as connection speed and port number in the case of Modbus).

3. Running programs created in graphical languages

CPDev engineering environment allows to create program organization units in all languages defined in IEC 61131-3 standard, both textual and graphical (Jamro *et al.* 2011a, b). The group of textual languages contains ST (*Structured Text*) and IL (*Instruction List*). FBD (*Function Block Diagram*), LD (*Ladder Diagram*), and SFC (*Sequential Function Chart*) are in the group of graphical languages. It is worth to mention that SFC is not considered as an independent language and it is also known as a mixed language, because it is required to prepare parts of the program in other languages of IEC 61131-3, i.e. ST, IL, FBD, or LD.

An interesting feature of the CPDev environment seems to be the debugging of graphical programs. All editors of graphical languages (i.e. FBD, LD, and SFC) are equipped

with an *execution mode*, that allows us to trace variable values and present them directly on the diagram. The most important features of the execution mode consist of:

- starting and suspending program execution,
- the presentation of variable values on the diagram (including values of connections between elements),
- changing values of input variables (forcing),
- saving values of the chosen variables in the text file,
- breakpoints support, both unconditional and conditional.

The details of the variable presentation and breakpoint support are explained in the next points of this paper.

3.1. Values presentation on diagrams

The presentation of current values on the diagram depends on the language and data. In the case of the FBD and LD editors, connections related to logic signals (BOOL) are drawn as a solid (TRUE) or dashed (FALSE) line. What is more, after pointing at function block input, output, or connection, an additional hint with a current variable value (related to the pointed element) is presented. An option of continuous presentation of the values is also available, which shows values directly above the connection lines.

Figure 2 presents SFC diagram in the execution mode. A special color is used to indicate an active step (Fig. 2, *SYSTEM_ON*). In an additional window, named *Variables*, the current values of the global and local variables declared in the program are presented. It is possible to change their values, which is a suitable solution during simulation. The user can test a program behavior when various input signals are available, without the necessity of obtaining data from real devices (e.g. sensors).

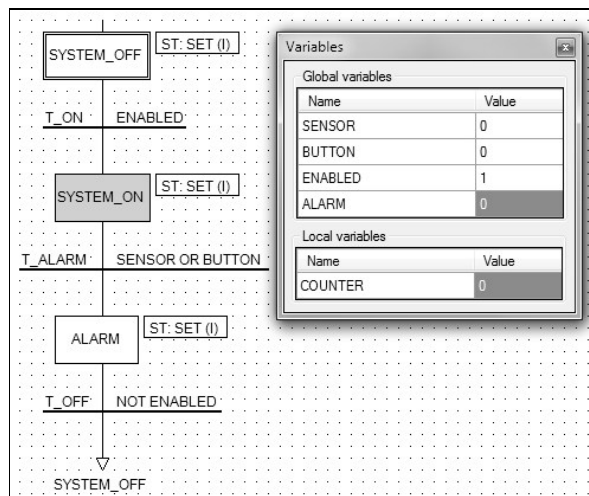

Fig. 2. An execution mode available in SFC editor

3.2. Breakpoints

Graphic editors are equipped with the possibility to stop program execution when software breakpoints are hit. They can be placed by the user next to various diagram elements, including functions, function block instances (FBD), coils (LD), and steps (SFC) (Fig. 3). Breakpoints are represented by small circles placed next to diagram elements. When a breakpoint is hit, program execution is stopped and the user can check system status or analyze variable values to ensure that the process is running as expected.

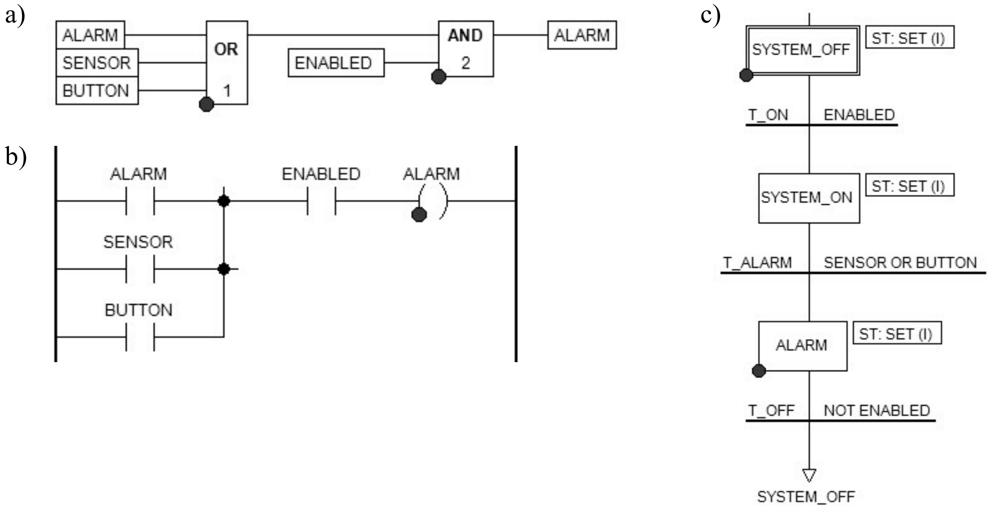

Fig. 3. Breakpoints on diagrams created in (a) FBD; (b) LD; (c) SFC

Both unconditional and conditional breakpoints are available. The first kind stops a program before executing an instruction related to the element with the breakpoint, e.g. before a function block call. This solution is sometimes uncomfortable for the user because it can cause the program to stop on each cycle of the program execution. To avoid this, conditional breakpoints are available, that stop program execution only when an additional condition is met. The condition is defined during breakpoint creation as an expression returning BOOL value. When a breakpoint is hit, program execution is stopped and the user can decide whether CPDev should ignore further breakpoint instances (Fig. 4).

As said, programs run under the control of the CPDev virtual machine. Additional virtual machine instruction *SCNTF* is used to indicate a place where a breakpoint is located. It is placed automatically in the code generated from the FBD, LD, or SFC diagram and then passed to the machine. Code 1 below presents an exemplary part of the code containing three breakpoints – two unconditional *SCNTF(1)*, *SCNTF(3)*, and one conditional – *SCNTF(2)*. The numbers passed to *SCNTF* are breakpoint identifiers. In the case of the conditional breakpoint 2, an additional IF instruction with a suitable conditional

(e.g. *SENSOR=TRUE*) is added. During program execution, when *SCNTF* is hit, the virtual machine suspends. The user can then check values of variables, and then resume program execution.

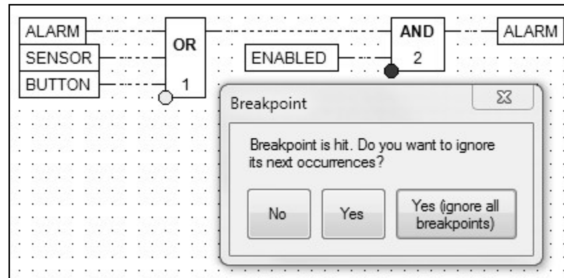

Fig. 4. Breakpoints during FBD program execution

Code 1

A part of generated code with breakpoints

```
SCNTF(1);
out_or1 := OR(ALARM, SENSOR, BUTTON);
IF SENSOR=TRUE THEN SCNTF(2); END_IF
out_and1 := AND(out_or1, ENABLED);
SCNTF(3);
ALARM := out_and1;
```

4. Tracing variable values in CPSim

CPSim is another module of CPDev programming environment that is used to test the execution of programs and to trace program variable values. It also uses the common mechanism of data sources, presented in p. 2. CPSim is especially useful at the final stages of project development, i.e. when the complete executable code for the virtual machine is created from all of the POUs. In a common scenario CPSim is used during the final on-line testing of the control system (commissioning).

The main window of the CPSim is presented in Figure 5. There is a project tree on the left where the following program resources are presented: global and local variables, tasks, programs, inputs, and outputs of function block instances. The user can connect with the selected data source (controller or simulator) and trace variable values by dragging them from the tree and placing them in the workspace. Figure 5 presents the workspace with two lists of variables on the left (global variables and inputs and outputs of *DELAY_ON* function block instance) and two panels (center) with a visual representation of BOOL variables (inputs *START*, *STOP*, *ALARM*, and outputs *ENGINE*, *PUMP*). Both the lists and the panels update their contents dynamically according to the current values of the monitored variables.

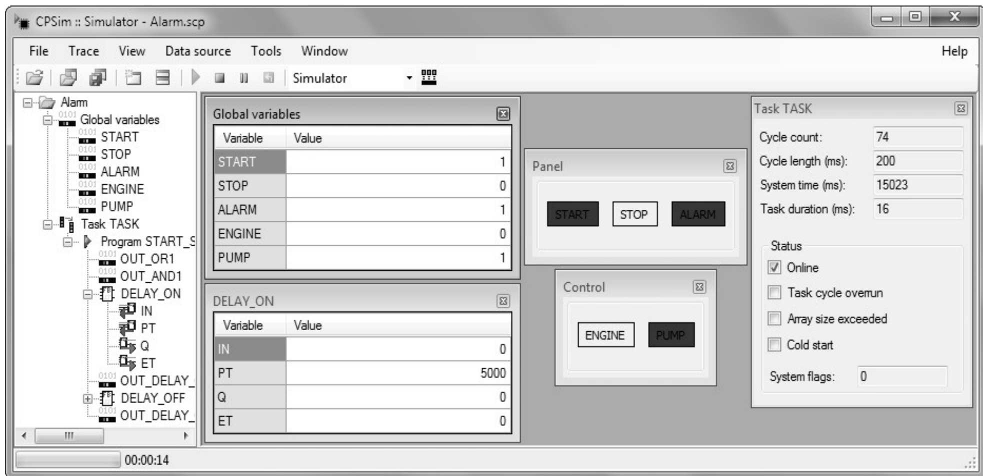

Fig. 5. CPSim main window

Apart from the variable values, task parameters can also be observed. In the case of Figure 5 there is only one project task *Task0001*. Parameters of the task are presented in the window on the right. The task runs every 200 ms, but its execution takes just 16 ms. The user can also monitor the number of executed cycles and the actual value of the controller system clock. CPSim presents situations which should require attention, e.g. violation of array size during program execution or overrunning of the cycle time.

The CPSim allows for the changing values directly in windows with variables. This can be used to test program response for various input values. The process can be automated by creation of special input file with values of input variables and information about a time when they should be set. The CPSim analyses them automatically and results are saved in the output file, that can be later used by external tools (e.g. Matlab, Excel). The program also saves all information about user actions and their outcomes. They are used to generate a report in HTML format for documentation purposes.

5. POU-oriented tests

During project development in the CPDev engineering environment a software engineer has the possibility to run verification tests oriented to the POUs (Program Organization Units). This is achieved with the CPTest application (Fig. 6) which main aim is to support two kinds of tests: unit and based on tables. The tests are executed with the usage of simulator (off-line mode) usually at the beginning of the software development, to ensure that the created POUs are verified before their usage in other system parts.

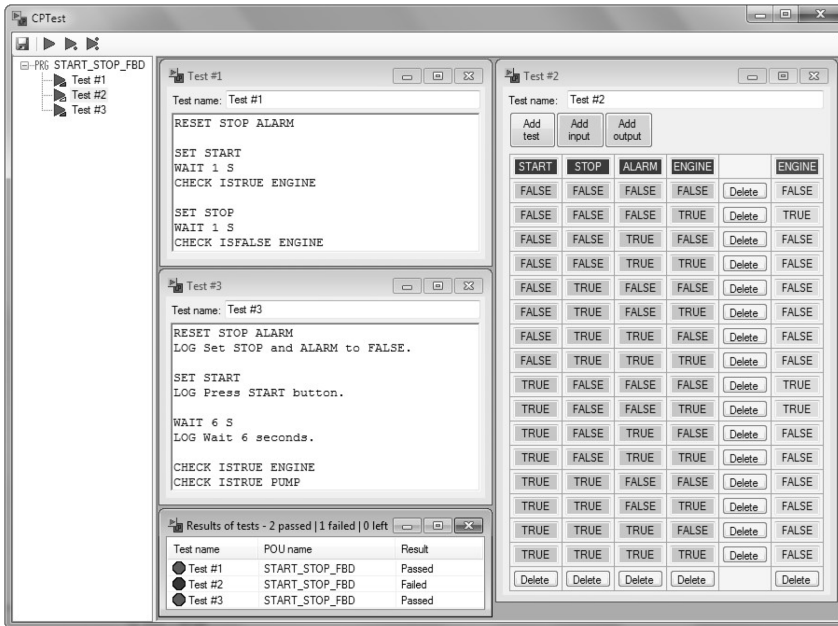

Fig. 6. CPTest main window

The CPTest application is equipped with a set of features to create and run unit tests, including:

- adding and removing tests,
- unit tests editor (Fig. 6, left part of the workspace),
- table tests editor (Fig. 6, right),
- running selected tests or all tests defined in the project, with the presentation of test run results (Fig. 6, bottom),
- saving and loading tests data in XML file.

5.1. Unit tests defined in CPTest language

The first group of tests supported by a CPTest consists of unit tests oriented to program organization units (POUs), i.e. programs, function blocks, and functions. It is especially important that they support creation of tests for function blocks and functions, because they are applied to the parts of code that are later reused. For example, POUs can be organized in libraries and then used in other CPDev projects.

Unit tests are created in a specially designed language, also called a CPTest, that contains the following instructions:

- *SET VARIABLE [VARIABLE ...]* – sets values of BOOL variable (or variables) to TRUE (e.g. *SET START ENGINE*),

- *RESET VARIABLE [VARIABLE ...]* – sets values of BOOL variable (or variables) to FALSE (e.g. *RESET START ENGINE*),
- *ASSIGN [VARIABLE] [VALUE]* – sets value of variable of a specific type (e.g. *ASSIGN COUNTER 100*),
- *WAIT [NUMBER] [UNIT]* – holds the test execution for a specified number of seconds (*S*), milliseconds (*MS*), minutes (*MIN*), and cycles (*C*); during this time program is still running, which allows us to check the values of variables after a defined period of time (e.g. *WAIT 5 S*, *WAIT 10 C*),
- *LOG [MESSAGE]* – saves a message in test run history (e.g. *LOG Starting test.*),
- *CHECK [OPERATOR] [VARIABLE] [VALUE]* – checks whether an expression consisting of arguments (separated by an operator) is correct (an incorrect expression means that test is failed); the following operators are supported: *EQ* (equal), *NEQ* (not equal), *LT* (less than), *LTE* (less than or equalto), *GT* (greater), and *GTE* (greater than or equal); an additional operators *ISTRUE* and *ISFALSE* are available, they require only one argument and allows us to check whether a BOOL variable is equal to TRUE (*ISTRUE*) or FALSE (*ISFALSE*) (e.g. *CHECK LT COUNTER 50, CHECK ISTRUE START*).

Unit tests are defined as a set of instructions presented above. Each of them has to be placed in a single line. Code 2a presents a sample test that checks whether *ENGINE* and *PUMP* variables are TRUE after 6 seconds since pressing the *START* button. Code 2b shows another unit test that can be used to make sure that 1s after pressing *START* button, *ENGINE* variable has the value TRUE, and later check whether the *ENGINE* variable has the value FALSE after 1s since pressing the *STOP* button.

Code 2

Sample unit tests created in the CPTest

(a)

```
01: RESET STOP ALARM
02: LOG Set STOP and ALARM to FALSE.

03: SET START
04: LOG Press START button.

05: WAIT 6 S
06: LOG Wait 6 seconds.

07: CHECK ISTRUE ENGINE
08: CHECK ISTRUE PUMP
```

(b)

```
01: RESET STOP ALARM

02: SET START
03: WAIT 1 S
04: CHECK ISTRUE ENGINE

05: SET STOP
06: WAIT 1 S
07: CHECK ISFALSE ENGINE
```

5.2. Tests based on tables

The second group of tests is based on tables. Its main advantage is no necessity of writing code to define a test, because the user only needs to fill values in the table. Table 1 contains an example of sucha test table. Its columns are divided into two parts: representing

input variables (on the left) and output variables (on the right). The table contains also a number of rows representing cases that will be checked by the CPTest application. For example, the first row causes the setting values of *START*, *STOP*, *ALARM*, and the *ENGINE* variables to FALSE, and then checks whether the *ENGINE* variable has a FALSE value.

The execution of these tests requires us to run them many times, force values of the variables (accordingly to the left part of the table), and check whether variables have exactly the same values as specified in the right part of the table. In the case of unequal results, the test fails. All of the information about the test run (including the values of the variables used and expected in a specific case) is stored for reference.

The table-based test mechanism makes it possible to add optional columns representing delays. They are useful to check values after a specific period of time.

Table 1
Exemplary tests table

START	STOP	ALARM	ENGINE	ENGINE
FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	**TRUE**	**TRUE**
FALSE	FALSE	**TRUE**	FALSE	FALSE
FALSE	FALSE	**TRUE**	**TRUE**	FALSE
FALSE	**TRUE**	FALSE	FALSE	FALSE
FALSE	**TRUE**	FALSE	**TRUE**	FALSE
FALSE	**TRUE**	**TRUE**	FALSE	FALSE
FALSE	**TRUE**	**TRUE**	**TRUE**	FALSE
TRUE	FALSE	FALSE	FALSE	**TRUE**
TRUE	FALSE	FALSE	**TRUE**	**TRUE**
TRUE	FALSE	**TRUE**	FALSE	FALSE
TRUE	FALSE	**TRUE**	**TRUE**	FALSE
TRUE	**TRUE**	FALSE	FALSE	FALSE
TRUE	**TRUE**	FALSE	**TRUE**	FALSE
TRUE	**TRUE**	**TRUE**	FALSE	FALSE
TRUE	**TRUE**	**TRUE**	**TRUE**	FALSE

6. Summary

The CPDev environment has been equipped with mechanisms which simplify testing and running the software created in IEC 61131-3 languages. The structure of these mechanisms is adjusted to language specificity (especially in the case of graphical languages) and applications, including control, monitoring and supervisory systems.

CPDev test tools can be used to improve project and code quality during the whole development process, starting from program debugging, POU-oriented unit tests, and up to the final commissioning tests. It is especially important due to the control and monitoring system's software complexity that is still being increased.

References

Jamro M., Sadolewski J. 2011a, *Edytor diagramów FBD jako moduł zintegrowanego środowiska CPDev* (in English: FBD editor as a module of CPDev integrated environment). [in:] Trybus L., Samolej S. (Eds): Projektowanie, Analiza i Implementacja Systemów Czasu Rzeczywistego. WKŁ, Warszawa.

Jamro M., Rzońca D., Sadolewski J., Stec A., Świder Z., Trybus B., Trybus L. 2011b, *Rozwój środowiska inżynierskiego CPDev do programowania systemów sterowania* (in English: Development of CPDev engineering environment for control system programming). [in:] Trybus L., Samolej S. (Eds): Projektowanie, Analiza i Implementacja Systemów Czasu Rzeczywistego. WKŁ, Warszawa.

Jamro M., Rzońca D., Sadolewski J., Stec A., Świder Z., Trybus B., Trybus L. 2011c, *Uruchamianie rozproszonego systemu kontrolno-pomiarowego* (in English: Execution of the distributed control and measurements system). [in:] Malinowski K., Dindorf R. (Eds): Postępy automatyki i robotyki cz. 1, Monografie t. 16, Komitet Automatyki i Robotyki Polskiej Akademii Nauk, Wydawnictwo Politechniki Świętokrzyskiej, Kielce, pp. 168–181.

IEC 61131-3 – Programmable Controllers. Part 3: Programming Languages.

Stec A., Świder Z., Trybus L. 2007, *Charakterystyka funkcjonalna prototypowego system do programowania systemów wbudowanych według normy IEC 61131-3* (in English: Functional specification of the prototype system for programming embedded systems according to IEC 61131-3). [in:] Huzar Z., Mazur Z. (Eds): Systemy Czasu Rzeczywistego. Metody i zastosowania. WKŁ, Warszawa.

Trybus B. 2011, *Development and Implementation of IEC 61131-3 Virtual Machine*, Theoretical and Applied Informatics, Vol. 23, No. 1/2011.

Hajduk Z., Sadolewski J., Trybus B. 2011, *FPGA-based Execution Platform for IEC 61131-3 Control Software*, Przegląd Elektrotechniczny (Electrical Review), R. 87, No. 8/2011, pp. 187–191.