Online ISSN 1848-3380, Print ISSN 0005-1144
ATKAFF 56(4), 499–507(2015)

Marcin Jamro, Dariusz Rzonca

Impact of Communication Timeouts on Meeting Functional Requirements for IEC 61131-3 Distributed Control Systems

DOI 10.7305/automatika.2016.01.838
UDK 681.518.5:004.414.22

Original scientific paper

The control software is frequently used in various systems that perform important and responsible tasks in industry. During its development, it is crucial to ensure that the solution is created in a way consistent with assumptions and meets all functional requirements. One of important steps consists of testing particular software units, separated from the rest of system, using the off-line simulator. However, test results can be different in case of a fully-connected system when external factors, such as communication issues, should be also taken into account. In this paper, the authors present a concept of specification and execution of system tests, using the dedicated test definition language, named CPTest+. It has been extended by the additional `ASSERT_COM` instruction, which performs an assertion that is able to detect problems related to external factors, including communication. To enable automatic and systematic testing, the dedicated metric has been proposed. It takes into account the current link status and archived results to calculate the probability that the test case has failed due to communication problems.

Key words: Control Systems, Communication, IEC 61131-3, Requirements, Testing

Utjecaj komunikacijskih prekida na zadovoljenje funkcionalnih zahtjeva IEC 61131-3 distribuiranih sustava upravljanja. Upravljački software je često korišten u sustavima koji obavljaju kritične zadatke u industriji. Tijekom njegovog razvoja, važno je osigurati zadovoljenje svih bitnih pretpostavki i funkcionalnih zahtjeva. Jedan od važnih koraka u razvoju navedenog softwarea je testiranje njegovih pojedinih cjelina korištenjem *offline* simulatora, neovisno o ostatku sustava. Međutim, rezultati testiranja mogu se bitnije razlikovati u slučaju potpuno povezanog sustava sa svim pripadajućim eksternim faktorima, kao što su komunikacijski problemi, koje je također potrebno uzeti u obzir. U ovom radu autori predstavljaju koncept definiranja i izvršenja testova sustava korištenjem jezika za namjensko definiranje testova, nazvanog CPTest+. On je proširen dodatnom `ASSERT_COM` naredbom koja obavlja provjeru s mogućnošću detekcije problema uzrokovanih vanjskim faktorima, npr. komunikacijom. Kako bi se omogućilo automatsko i sistematično testiranje predložen je namjenski mjerni sustav. On uzima u obzir trenutni status veze kao i arhivirane rezultate kako bi se odredila vjerojatnost neizvršenja testnog slučaja uslijed komunikacijskih problema.

Ključne riječi: sustavi upravljanja, komunikacija, IEC 61131-3, zahtjevi, testiranje

1 INTRODUCTION

The control software is an important subclass of the real-time software and is often used in industry, such as in Programmable Logic Controllers (PLCs), Programmable Automation Controllers (PACs), and Distributed Control Systems (DCSs). Due to its specific requirements, it is important to prepare such a kind of software in a way that ensures its high quality and operation in erroneous scenarios, as well as decreases a number of problems that may be found while running on the plant. As stated in [1], there are some possibilities that could be useful for improving quality of control software, such as its modeling, standardized implementation, and detailed testing.

The concept of using various kinds of tests for detecting potential problems in control software is becoming more and more popular. One of approaches is based on the idea of unit testing [2] that checks whether small parts of implementation meet requirements. Apart from unit tests, the system should be verified using a set of integration and system tests.

A subject of software testing is a complex and difficult problem that may be also connected with the Test-Driven Development (TDD) paradigm. It is based on the assumption that a developer prepares tests for a particular feature before writing the implementation code. Then, the developer works on the code until all tests are passed. The last phase is named refactoring and allows to improve the code quality, but it does not change the external behavior of a

unit [3]. The TDD approach has many advantages, such as creating code with a smaller number of functional errors [4], as well as limiting amount of time necessary for debugging [5].

In case of control software created according to the IEC 61131-3 standard [6], a single Program Organization Unit (POU, namely program, function block, function, or class) can be understood as a unit for testing. There are already some solutions that allow to create and execute tests oriented towards POUs. One of them is shown in [7] and describes a dedicated test definition language, named CPTest+. It has been also extended for testing performance of particular POUs [1] and communication between devices in DCSs [8]. Other interesting approaches to control software testing include the test-driven automation concept [9], industrial automation software development process based on tests [10], agile keyword-driven testing method [11], and deterministic replay debugging [12].

Testing of POUs separated from environment is crucial for creation of a well-tested solution. However, it is also very important to check how the system behaves when all its parts are taken into account. For this reason, the testing team should also prepare a set of system tests. They are often used to verify that functional requirements are met when all system modules are prepared. Nevertheless, such a problem is a bit different in case of DCSs. Typically, such systems [13] consist of various devices exchanging data through an industrial network [14]. A scale of the system may vary. In a trivial case, mini-DCS involves a single controller communicating with remote I/O modules or HMI panel using a simple fieldbus [15], but large applications are fairly common. Even in mini-DCS communication errors or delays may lead to performing calculations on invalid or out-of-date values. Unfortunately, a real error cause may not be obvious to determine, especially during automated test procedures.

In this paper, an attempt has been made to propose a solution for estimating communication impact on satisfying functional requirements of control software. Requirements and parameters of communication are specified in the Systems Modeling Language (SysML) [16]. The existing CPTest+ language has been enhanced to handle a new kind of assertion, namely `ASSERT_COM` that deals with communication issues during testing. Such an assertion tries to automatically distinguish between errors in control program implementation and these related to external factors, such as communication flaws. To achieve this goal, the authors propose an additional function for calculating probability that a particular test case has failed due to communication problems.

The approach could be used with the CPDev engineering environment[1] [17] developed in the Department of Computer and Control Engineering at Rzeszow University of Technology (Poland). It consists of several parts, cooperating together and forming the comprehensive solution for modeling, implementation, testing, configuration, simulation, visualization, and commissioning of control software for various controllers. The environment has a few industrial applications, for instance in ship control and monitoring system developed by Praxis Automation Technology B.V.[2] from the Netherlands. CPDev could be also used for programming softPLCs and the fast FPGA controller.

The paper is organized as follows. The second section presents an existing concept of testing with the CPTest+ language, equipped with the modeling features using the SysML language. In the next section, the assertion-based extension, introducing the `ASSERT_COM` instruction, is shown. The fourth section describes a way of results analysis, together with explanation of the dedicated metric. The laboratory stand and example of the approach are presented in the fifth section.

2 TESTING CONCEPT WITH MODELING

The control software needs to satisfy various requirements, including functional ones. There are already some possibilities of testing whether such requirements are met. One of variants uses the unit testing concept to verify whether POUs are working properly. A solution dedicated to IEC 61131-3 projects has been described in [7]. It supports the project structure shown in Fig. 1. The control system contains a set of resources. Each of them executes one or more tasks. Each task contains one or more POUs, which have various tests assigned.

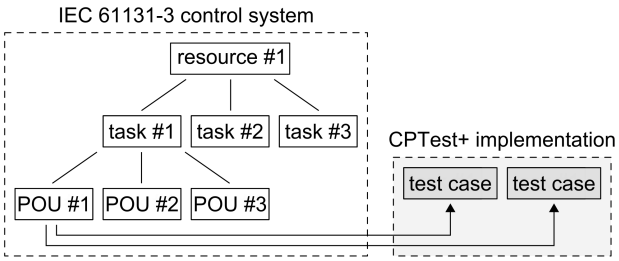

Fig. 1: Assignment of CPTest+ test cases to POUs

2.1 CPTest+ Dedicated Test Definition Language

The dedicated test definition language, named CPTest+, has been proposed to develop test cases. In the basic version, presented in [18], the language supports a few instructions that allow to perform various operations related to manual testing, such as setting a value of

[1] `http://cpdev.kia.prz.edu.pl/`

[2] `http://praxis-automation.com/`

variable (SET, RESET, ASSIGN), holding execution of the test case for a given period of time (WAIT), saving additional information to the test run log (LOG), as well as verifying whether the current value of variable is consistent with expected, using a given operator (ASSERT with various operators, such as EQ or LT).

It is worth mentioning that the CPTest+ language can be expanded to support some features specific to a particular test type, such as communication performance tests [19] or POU-oriented performance tests [1]. What is more, CPTest+ can be used to develop not only unit tests, but also integration and system ones, as shown in the following part of this paper.

2.2 SysML-based Modeling

To provide engineers with a convenient way of developing and maintaining control system projects that use the IEC 61131-3 languages, that is beneficial to propose a suitable modeling approach. It could support the Model-Driven Development (MDD) concept [20], which focuses on the model as the most important artifact in the development process. Stored model data could be used to generate implementation or its parts in an automatic way. Such an approach has several applications in various software development domains, including real-time embedded systems [21].

In this paper, the authors present a part of the modeling methodology that allows to design POUs and tests, as well as assign tests to particular POUs. The concept uses the SysML language [16], which is an extension to Unified Modeling Language (UML) [22], and provides engineers with nine diagram types for presentation of structure, behavior, and requirements. The modeling approach is based on the preliminary version described in [23].

Each POU is shown on a separate Block Definition Diagram (BDD), located in the POUs package. A single POU is represented as a block marked with two stereotypes. The first represents the POU type and can be chosen from the following: «*program*», «*functionBlock*», «*function*», and «*class*». The second stereotype indicates a language that will be used for implementation. It can be set as «*st*» (Structured Text), «*il*» (Instruction List), «*fbd*» (Function Block Diagram), «*ld*» (Ladder Diagram), or «*sfc*» (Sequential Function Chart). A name of the block is used as a name of the POU. Each block may also contain a set of proxy ports that represent inputs or outputs, depending on a port direction.

In the example from Fig. 2, the SENSOR function block is defined. It contains four inputs (ENABLED of the BOOL type, as well as VALUE, MIN_VALUE, and MAX_VALUE of REAL) and the STATE output (of INT). To clarify constructions placed on diagrams, the comment is added.

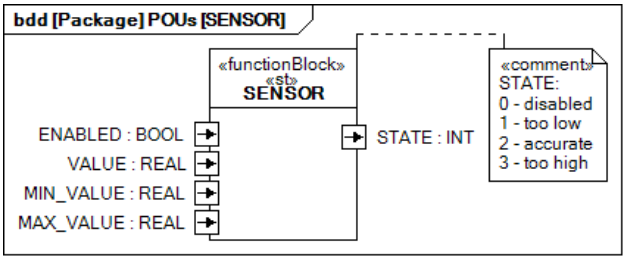

Fig. 2: Modeling of the SENSOR function block

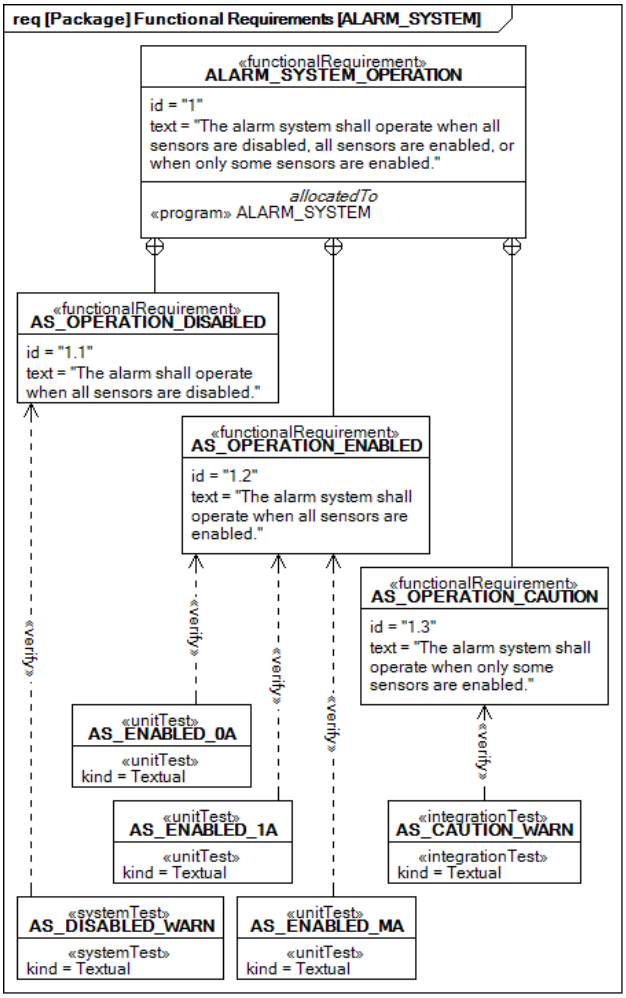

Fig. 3: Modeling of POU functional requirements

The presented part of the modeling methodology does not define tests nor their assignment to particular POUs. For this reason, the approach supports specification of functional requirements for a given POU on the Requirement Diagram (REQ), as shown in Fig. 3. It has many similarities with the overall approach to modeling func-

tional and nonfunctional requirements, described in details in [24]. For the purpose of convenient modeling of various test kinds, three stereotypes are defined. They are specialization of «*testcase*» and are named «*unitTest*», «*integrationTest*», and «*systemTest*». The first indicates a unit test, the second – integration, while the other – a system one. The REQ diagram contains the main «*functionalRequirement*» element that represents a general functional requirement for a given POU. It has a unique identifier (`id`), a description (`text`), as well as allocation to a particular POU (`allocatedTo` compartment). This block can be connected with other «*functionalRequirement*» blocks by the containment relationship. Each of them should also contain a unique identifier and a description, however, assignment to a POU is unnecessary, because it is provided by the parent requirement. Tests are created to verify whether requirements are satisfied. Thus, «*unitTest*», «*integrationTest*», or «*systemTest*» elements are placed on the diagram. They are connected with suitable requirements by «*verify*» relationships. Each test is parameterized by a name and a kind. For instance, the `Textual` kind indicates that CPTest+ is used for test implementation.

As an example, the REQ diagram from Fig. 3 specifies requirements for the `ALARM_SYSTEM` program. The diagram contains the general requirement, named `ALARM_SYSTEM_OPERATION`, with three subrequirements. They present functional requirements for the program in three scenarios – (1) when all motion sensors are disabled, (2) when all sensors are enabled, and (3) when only some sensors are enabled. Such requirements are verified using five tests – a system one for the first (`AS_DISABLED_WARNING`), three unit for the second (`AS_ENABLED_0A`, `AS_ENABLED_1A`, and `AS_ENABLED_MA`), as well as an integration one for the other (`AS_CAUTION_WARNING`). All of them will be created in the CPTest+ test definition language, as specified by a value of the `kind` property.

3 ASSERTION-BASED EXTENSION

For testing impact of communication timeouts on meeting functional requirements, the CPTest+ language has been extended with the `ASSERT_COM` instruction, which operation is presented in Alg. 1. Its main aim is to extend the "classic" assertion by checking whether incorrect results are caused by functional errors in POU implementation or by external problems, such as with communication.

At the beginning (line 1 in Alg. 1), the instruction checks whether the "classic" assertion is satisfied, i.e., the actual value is consistent with the expected one. It takes into account the operator, such as equality (`EQ`), inequality (`NEQ`), as well as relations (`LT`, `LTE`, `GT`, `GTE`). Such an instruction also supports two special operators related to logical values, namely `ISTRUE` and `ISFALSE`. These two operators check whether the value is equal to `TRUE` and `FALSE`, respectively. Regarding the overall concept for all operators, the actual and expected values are combined into the expression, using the operator. The exemplary expression is $NUMBER > 2$, where the actual value is a value of the `NUMBER` variable, the operator is the "greater" relation (`GT`), and the expected value is equal to `2`. If the "classic" assertion is satisfied, the test proceeds to the following instruction or till the end.

Algorithm 1 Operation of the `ASSERT_COM` instruction

Input: $condition$ is a boolean expression associated with the assertion

```
1.  if condition = true then
2.      continue execution of the test {the "classic" assertion is met}
3.  else
4.      create a local copy of values of all variables
5.      if an error in the communication is found then
6.          get a cycle time for the current task
7.          repeat
8.              create a local copy of values of all variables
9.              hold test execution for the cycle time
10.         until the "classic" assertion is met or the maximum waiting time elapsed
11.     else
12.         fail the test due to functional problems
13.     end if
14. end if
```

Behavior of the `ASSERT_COM` instruction differs significantly from `ASSERT` when the "classic" assertion is failed. In such a scenario, the mechanism creates a local copy of values of all variables (line 4). Such data are stored in the special form and are associated with the test run log. They can be later used for analysis and debugging.

Then, the system checks whether any error is found in communication, by reading a value of the special indicator from the controller (line 5). It provides information whether a problem could be caused by communication problems, such as a necessity of retransmission.

If the `ASSERT_COM` instruction discovers that the current error may be caused by communication problems, a few actions are executed in the loop (lines 7–10). It can be stopped only if the "classic" assertion is met or when the maximum waiting time elapsed. In each iteration, the system creates a local copy of values of all variables, as well as waits the cycle time. Such a construction allows to check whether the "classic" assertion can be passed in one of the closest cycles. In such a situation, it is possible that the result is affected by communication problems.

Of course, during the consecutive checks, the system under test is still running. Thus, passing the assertion in one of the following cycles could be caused not only by the proper communication, but also by behavior of the control program. For this reason, the ASSERT_COM instruction is not suitable for all test scenarios.

Calling the ASSERT_COM instruction in the test is very similar to using ASSERT, because only a name of the instruction differs, while parameters remain the same. The code of an exemplary system test is shown as follows:

```
01: SET READ_TEMPERATURE
02: WAIT 1 C
03: ASSERT_COM NEQ S1_VALUE 0
04: ASSERT_COM NEQ S2_VALUE 0
05: ASSERT_COM NEQ S3_VALUE 0
06: WAIT 1 C
07: ASSERT_COM EQ S1_STATE 3
08: ASSERT_COM EQ S2_STATE 2
09: ASSERT_COM EQ S3_STATE 1
```

The above test should be run in the laboratory with a configured testing environment, where sensors can read proper temperature values – too high by the first sensor, accurate by the second, and too low for the other. It is assumed that values read from physical sensors are stored to S1_VALUE, S2_VALUE, and S3_VALUE global variables, while calculated states are stored to S1_STATE, S2_STATE, and S3_STATE. The test case checks whether values are read correctly from sensors within one cycle (line 2) after setting a value of the READ_TEMPERATURE global variable (line 1). The test assumes that the additional monitoring program zeros values of global variables representing temperature values obtained from sensors, if READ_TEMPERATURE is equal to FALSE. With the usage of the presented test, it is possible to detect situations when values are not read correctly from sensors within a given cycle time (lines 3–5), what could have an impact on the system future operations by using out-of-date values, such as in lines 7–9.

4 RESULTS ANALYSIS

During execution of tests equipped with at least one ASSERT_COM instruction, a lot of data are obtained and stored. It is especially well visible in case of complex projects, where operations are performed on hundreds or even thousands of variables. For this reason, the authors propose a mechanism of analyzing gathered results, which uses data associated with the test run log to present values of variables as charts (Fig. 4). They may contain indicators representing a moment when the "classic" assertion is failed, passed, or when the timeout occurred.

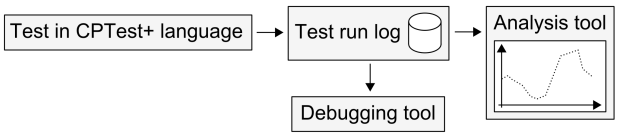

Fig. 4: Using data from the test run log for results analysis

The generated charts can be used to manually check whether the problem with passing functional requirement is caused by an external factor, such as a communication problem. However, such a solution can be cumbersome and does not support an idea of automated testing. Thus, the authors propose a dedicated function that informs an engineer about a probability that the error is related to communication problem. It takes values from range $[0; 1]$ and can be calculated as:

$$m_{\mathrm{ci}}\left(p_{\mathrm{lst}}, c\right) = \begin{cases} 0 & \text{if } p_{\mathrm{lst}} \in [0; 0.1\psi] \\ \delta & \text{if } p_{\mathrm{lst}} \in (0.1\psi; 0.9\psi) \wedge c \in [1; \gamma] \\ 0 & \text{if } p_{\mathrm{lst}} \in (0.1\psi; 0.9\psi) \wedge c \in (\gamma; \infty) \\ 1 & \text{if } p_{\mathrm{lst}} \in [0.9\psi; \psi] \end{cases}$$

$$\delta = \beta + \frac{(\alpha - \beta)(c - 1)}{\gamma - 1}$$

The γ parameter is a number of cycles taken into account to check whether the problem is caused by communication issues, while p_{lst} indicates how many tests from ψ last have been passed, since the last change in implementation of POUs involved in the given test. Parameters α and β are minimum and maximum values of m_{ci}, respectively, when $p_{\mathrm{lst}} \in (0.1\psi; 0.9\psi)$. The c ($cycles$) parameter represents a number of cycles in the current test run that were necessary to pass functional requirements. Of course, parameters $\alpha, \beta \in [0; 1]$, $\alpha < \beta$, $\gamma \in \mathbb{N}_+$, $\psi \in \mathbb{N}_+$, while arguments $p_{\mathrm{lst}} \in \mathbb{N}$, $p_{\mathrm{lst}} \leqslant \psi$, and $c \in \mathbb{N}_+$.

At the beginning, it has been assumed that the probability of causing problems by communication decreases linearly with increasing number of necessary cycles. However, other functions for δ may be considered, as well. Thus, modeling of the non-linear change is also possible, such as according to the modified formula:

$$\delta = \beta + \frac{(\alpha - \beta)(c - 1)^n}{(\gamma - 1)^n}$$

The additional parameter n may be set as a value from $[1; \infty)$. Thus, previous simplified equation may be seen as a special case where $n = 1$.

In case of the exemplary application, the authors propose to use the following values for parameters: $\alpha = 0.1$, $\beta = 0.9$, $\psi = 10$. Thus, if the test has been passed in at least 9 of the last 10 runs, it is highly possible that the

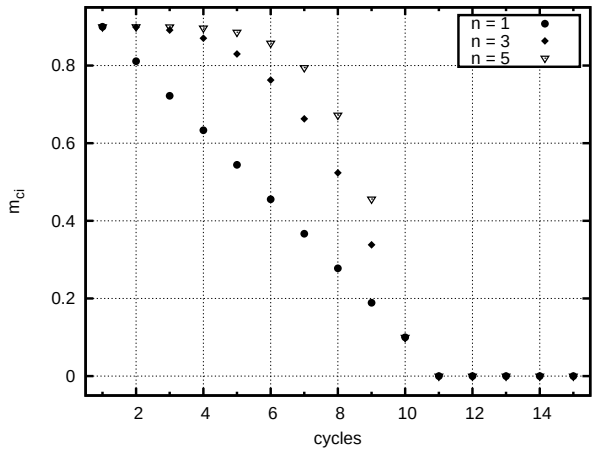

Fig. 5: Value of the m_{ci} for $p_{\mathrm{lst}} \in (0.1\psi; 0.9\psi)$, $\alpha = 0.1$, $\beta = 0.9$, $\gamma = 10$, $\psi = 10$.

problem is related to external factors, such as communication. It is an acceptable assumption, because tests should be prepared in a way to be repeatable and isolated from the system environment. Otherwise, it is more possible that the problem is caused by implementation. The authors assume that with increasing number of cycles, the probability of causing problems by communication decreases. Such an assumption is visible in the equation, as well as in the chart shown in Fig. 5.

By using the dedicated metric, the concept of measuring impact of communication timeouts on meeting requirements can be integrated with the automated testing mechanism. The result of tests, using the `ASSERT_COM` instruction, can take one out of three verdicts:

- passed, if the „classic" assertion is met,
- failed, if $m_{\mathrm{ci}} < 0.5$, i.e., there is probably an error in implementation,
- undecidable, if $m_{\mathrm{ci}} \geqslant 0.5$, i.e., there is probably an error in communication.

As shown above, the approach requires to introduce a new verdict, namely *undecidable*. It is important from the perspective of this paper, because results obtained with external problems (such as communication) cannot be used as a reliable source of information regarding the system correctness and robustness. Nevertheless, it also does not mean that the implementation is incorrect, thus using the *failed* verdict instead is unsuitable in this scenario.

5 LABORATORY STAND

To check the practical usefulness of the described approach, the authors have built the simple laboratory stand. This exemplary application is shown in Fig. 6.

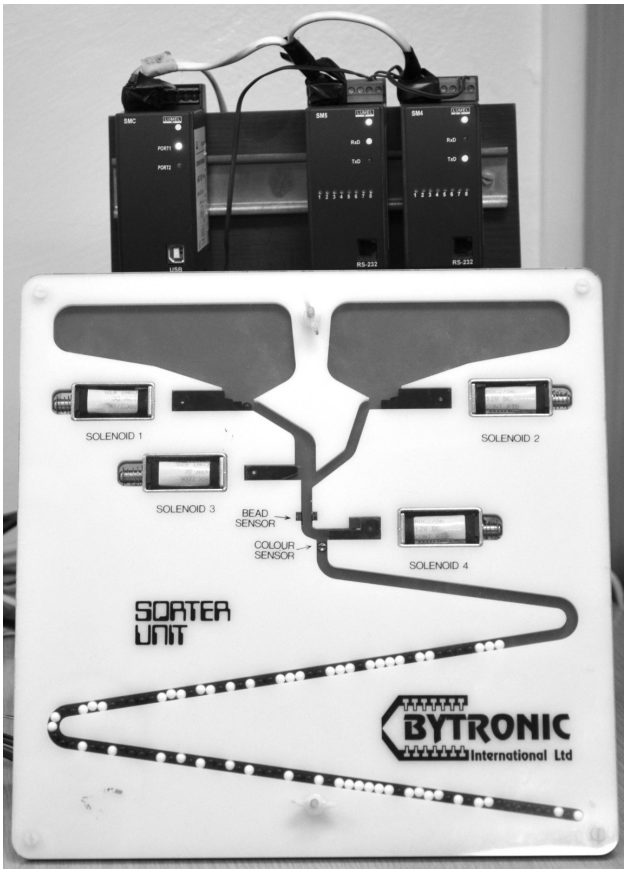

Fig. 6: Laboratory stand

The prototype laboratory stand consists of the PC with the CPDev engineering environment and the CPTest application, the SMC controller equipped with SM4 and SM5 external I/O modules[3], as well as the Bytronic Sorter Unit (BSU)[4]. The BSU may function as a sorting machine, where black and white beads should be appropriately separated to two hoppers, or alternatively to produce different patterns in the zig-zag track from the beads consecutively dispensed from the hoppers.

In the current example, the latter case is used. Dispensing the beads from the hoppers is controlled by two gates (one for each hopper), opened and closed by appropriate solenoids, connected to the binary outputs module (SM4). The gate should be open only for a short period of time to ensure that only one bead has fallen. The infrared beam sensor connected to the binary input module (SM5) is used as a feedback to confirm that the bead has been successfully dispensed.

A part of requirements for the pattern generator, cre-

[3] `http://www.lumel.com.pl/en/`
[4] `http://bytronic.net/html/sr.html`

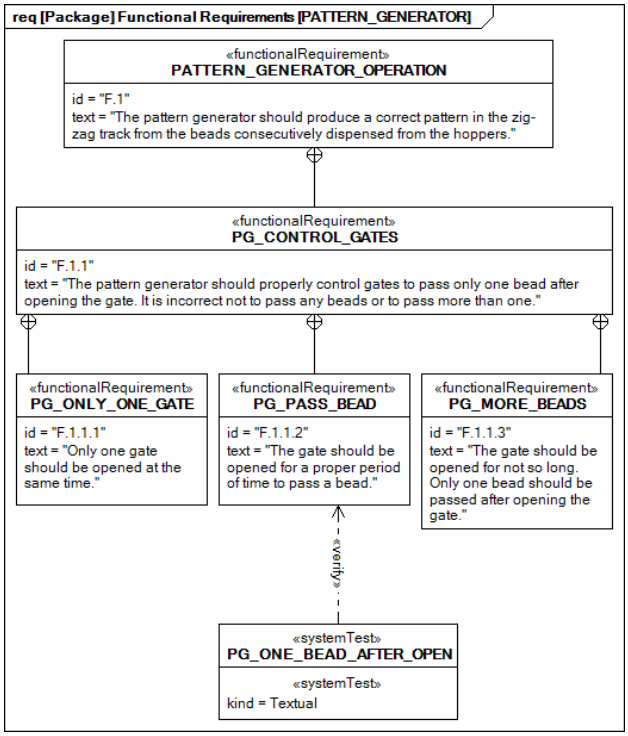

Fig. 7: Modeling of requirements for the exemplary system

ated in the SysML graphical modeling language, is presented in Fig. 7. The diagram defines five requirements organized in a three-level hierarchy, as well as the PG_ONE_BEAD_AFTER_OPEN test. Such a test verifies the PG_PASS_BEAD requirement and is explained in the following part of this section. To simplify the example, other requirements do not have any test cases assigned.

Such a system is very small and simple, however, it could be used to show a real test scenario where the proposed testing approach is applicable. Absence of signal from the bead sensor may be caused by numerous reasons, both functional and non-functional ones. It could be triggered by an error in the program code (such as opening the gate for too short) or external reasons (such as an empty hopper or a communication error between the SMC controller and one of the I/O modules). Automatic distinguishment between those cases is very helpful, especially during the regression testing. Thus, the following test in the CPTest+ dedicated test definition language could be used:

```
01: SET PASS_ONE_BALL
02: LOG ONE BALL MODE ON
03: WAIT 500 MS
04: RESET PASS_ONE_BALL
05: LOG ONE BALL MODE OFF
06: ASSERT_COM ISTRUE IS_BEAD_SEEN
```

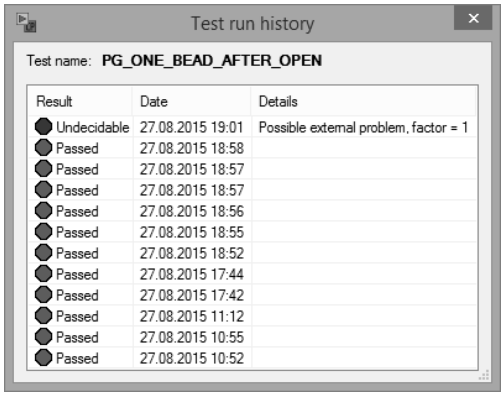

Fig. 8: Results presented in the CPTest tool

The test checks whether the first gate is open for not too short period of time to pass a bead. The presented test case starts with setting a value of the PASS_ONE_BALL global variable to TRUE (line 1). It is handled by the control software and indicates that the gate should be open for a suitable period of time just to pass one bead. Such a process requires to communicate twice with the binary outputs module – first to open the gate, and after a few tens of milliseconds once again to close it. Then, information about the current status is stored in the test run log (line 2), and the test execution is held for 500 milliseconds (line 3). Of course, when the test execution is held, the control software still runs to pass the bead. The PASS_ONE_BALL variable is reset after 500 milliseconds (line 4) and another message is logged (line 5). At the end, the ASSERT_COM instruction is applied to check whether a bead has been discovered by the bead sensor. The global variable IS_BEAD_SEEN is used. It is set by the control software, according to results obtained from the binary input module. For this reason, the assertion allows to find various kinds of problems, including an error in the implementation, as well as problems with communication.

As an example of analysis of the measured results, the experiment in one hypothetical case will be considered. Let assume that the control program is correct (i.e., the gate is opened for sufficient time to pass one bead), but after several successful test runs, an external error is intentionally introduced (such as a communication cable is disconnected). In such an experiment, the test run log shows that the assertion has been passed numerous times, however, finally it has been hit with verdict suggesting that the reason is external, with probability m_{ci}. Such an experiment result could be presented in the CPTest window (Fig. 8).

6 CONCLUSION

The quality of control software, created according to the IEC 61131-3 standard, can be improved in a few ways,

such as by precise testing. Apart from checking whether particular POUs operate correctly in separation from other components, it is crucial to ensure that the whole system operates as planned. Such a task is very important in case of DCSs where problems with communication between devices may cause unexpected software operation.

In the paper, the enhanced version of tests created in the CPTest+ dedicated test definition language have been proposed. An influence of communication problems on meeting functional requirements for the currently tested unit has been minimized, due to the dedicated `ASSERT_COM` instruction. The special assertion can be repeated many times to check whether in the following cycles the requirement is satisfied. What is more, the dedicated function has been proposed to identify a source of error by distinguishing between problems related and unrelated to external factors. The proposed approach also promotes analysis of data collected during such tests, because they reveal behavior of the system in real imperfect environment.

REFERENCES

[1] M. Jamro, "Development and Execution of POU-oriented Performance Tests for IEC 61131-3 Control Software," in *Recent Advances in Automation, Robotics and Measuring Techniques* (R. Szewczyk, C. Zielinski, and M. Kaliczynska, eds.), vol. 267 of *Advances in Intelligent Systems and Computing*, pp. 91–101, Springer, 2014.

[2] J. Dooley, "Unit Testing," in *Software Development and Professional Practice*, pp. 193–208, Apress, 2011.

[3] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making Program Refactoring Safer," *Software, IEEE*, vol. 27, no. 4, pp. 52–57, 2010.

[4] L. Crispin, "Driving Software Quality: How Test-Driven Development Impacts Software Quality," *Software, IEEE*, vol. 23, no. 6, pp. 70–71, 2006.

[5] J. Grenning, "Applying test driven development to embedded software," *Instrumentation Measurement Magazine, IEEE*, vol. 10, no. 6, pp. 20–25, 2007.

[6] "IEC 61131-3 - Programmable controllers - Part 3: Programming languages," 2013.

[7] M. Jamro, "POU-oriented Unit Testing of IEC 61131-3 Control Software," *Industrial Informatics, IEEE Transactions on*, 2015. accepted.

[8] M. Jamro, D. Rzonca, and W. Rząsa, "Testing communication tasks in distributed control systems with SysML and Timed Colored Petri Nets model," *Computers in Industry*, vol. 71, pp. 77–87, 2015.

[9] D. Winkler, R. Hametner, and S. Biffl, "Automation component aspects for efficient unit testing," in *Emerging Technologies Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pp. 1–8, 2009.

[10] R. Hametner, D. Winkler, T. Ostreicher, S. Biffl, and A. Zoitl, "The adaptation of test-driven software processes to industrial automation engineering," in *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pp. 921–927, 2010.

[11] R. Hametner, D. Winkler, and A. Zoitl, "Agile testing concepts based on keyword-driven testing for industrial automation systems," in *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pp. 3727–3732, 2012.

[12] H. Prahofer, R. Schatz, C. Wirth, and H. Mossenbock, "A Comprehensive Solution for Deterministic Replay Debugging of SoftPLC Applications," *Industrial Informatics, IEEE Transactions on*, vol. 7, no. 4, pp. 641–651, 2011.

[13] L. Feng-Li, W. Moyne, and D. Tilbury, "Network design consideration for distributed control systems," *Control Systems Technology, IEEE Transactions on*, vol. 10, no. 2, pp. 297–307, 2002.

[14] P. Gaj, J. Jasperneite, and M. Felser, "Computer communication within industrial distributed environment – a survey," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 1, pp. 182–189, 2013.

[15] IEC, "IEC 61158 Standard: Industrial Communication Networks – Fieldbus Specifications," 2007.

[16] OMG, "OMG Systems Modeling Language, V1.3," 2012.

[17] M. Jamro, D. Rzonca, J. Sadolewski, A. Stec, Z. Swider, B. Trybus, and L. Trybus, "CPDev Engineering Environment for Modeling, Implementation, Testing, and Visualization of Control Software," in *Recent Advances in Automation, Robotics and Measuring Techniques* (R. Szewczyk, C. Zielinski, and M. Kaliczynska, eds.), vol. 267 of *Advances in Intelligent Systems and Computing*, pp. 81–90, Springer, 2014.

[18] M. Jamro and B. Trybus, "Testing Procedure for IEC 61131-3 Control Software," in *12th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems (PDeS)*, pp. 192–197, 2013.

[19] M. Jamro and D. Rzonca, "Measuring, Monitoring, and Analysis of Communication Transactions Performance in Distributed Control System," in *Computer Networks* (A. Kwiecien, P. Gaj, and P. Stera, eds.), vol. 431 of *Communications in Computer and Information Science*, pp. 147–156, Springer International Publishing, 2014.

[20] D. Hastbacka, T. Vepsalainen, and S. Kuikka, "Model-driven development of industrial process control applications," *The Journal of Systems and Software*, vol. 84, no. 7, pp. 1100–1113, 2011.

[21] J. Babic, S. Marijan, and I. Petrovic, "Introducing Model-Based Techniques into Development of Real-Time Embedded Applications," *Automatika – Journal for Control, Measurement, Electronics, Computing and Communications*, vol. 52, no. 4, pp. 329–338, 2011.

[22] OMG, "OMG Unified Modeling Language, Infrastructure, V2.4.1," 2011.

[23] M. Jamro and B. Trybus, "An approach to SysML modeling of IEC 61131-3 control software," in *Methods and Models in Automation and Robotics (MMAR), 2013 18th International Conference on*, pp. 217–222, 2013.

[24] M. Jamro, "SysML Modeling of Functional and Non-functional Requirements for IEC 61131-3 Control Systems," in *Progress in Automation, Robotics and Measuring Techniques* (R. Szewczyk, C. Zielinski, and M. Kaliczynska, eds.), vol. 350 of *Advances in Intelligent Systems and Computing*, pp. 91–100, Springer International Publishing, 2015.

Marcin Jamro received B.Sc., M.Sc. in computer engineering, as well as Ph.D. in computer science at Rzeszow University of Technology (Poland) in 2011, 2012, and 2015, respectively. He was a research assistant at Rzeszow University of Technology till 9/2015. His research focuses on software engineering of real-time systems, especially their modeling and testing. Author and co-author of more than 25 publications, including scientific papers, chapters in monographs, and a book.

Dariusz Rzonca received B.Sc. in mathematics at University of Rzeszow in 2002, M.Sc. in computer engineering at Rzeszow University of Technology in 2004, and Ph.D. in computer science at Silesian University of Technology in 2012. He has been working as an assistant professor at Rzeszow University of Technology. His research focuses on Petri nets, industrial control systems, embedded systems, and cryptography. Author and co-author of more than 50 publications.

AUTHORS' ADDRESSES

Marcin Jamro, Ph.D.
Dariusz Rzonca, Ph.D.
Department of Computer and Control Engineering,
Faculty of Electrical and Computer Engineering,
Rzeszow University of Technology,
al. Powstańców Warszawy 12, 35-959 Rzeszów, Poland
email: {mjamro, drzonca}@kia.prz.edu.pl

Received: 2014-04-16
Accepted: 2015-10-05