Using Process-Oriented Structured Text for IEC 61499 Function Block Specification

Vladimir Zyubin(✉) and Andrei Rozov

Institute of Automation and Electrometry, Acad. Koptyuga prosp. 1, 630090 Novosibirsk, Russia
{zyubin,rozov}@iae.nsk.su

Abstract. This paper deals with leveraging the IEC 61499 Function Blocks with the poST language. The poST language is a process-oriented extension of the IEC 61131-3 Structured Text (ST) language. The language targets specifying stateful behavior of PLC-based control software. The main purpose of our contribution is to provide coherence of the PLC source code with technological description of the plant operating procedure. The language combines the advantages of FSM-based programming with conventional syntax of the ST language and can be easily adopted by the community. The poST language assumes that a poST-program is a set of weakly connected concurrent processes. Each process is specified by a sequential set of states. The states are specified by a set of the ST constructs, extended by TIMEOUT operation, SET STATE operation, and START/STOP/check state operations to communicate with other processes. The paper describes the basics of the poST language, design constructs, and demonstrates usage of the poST language by developing control software for a wheel-chair elevator, and discusses the poST language over the control software implementation in Execution Control Chart. AQ1

Keywords: PLC languages · IEC 61499 · IEC 61131-3 · Control software development · Process-oriented programming · ECC

1 Introduction and Motivation

The ongoing transition to Industry 4.0 means a dramatic increase in complexity and use of embedded and cyber-physical systems in our lives. This demands a reassessment of the tools used for design and development of such systems. Behavior of a cyber-physical system is determined by the control system, and behaviour of control system is specified by software. Thus the models, methods and languages employed in development of control software need to be revised.

The majority of automation professionals predict that the tooling for Industry 4.0 software will be based on IEC 61131-3 and its evolution to IEC 61499.

This work was supported by the Russian Ministry of Education and Science, project no. AAAA-A19-119120290056-0.

There is a heated debate on the pros and cons of these standards in research papers. On the one hand, it is stated that IEC 61131-3 [1], developed in the early 90s and having known shortcomings, needs serious refactoring [2]. On the other hand, there are active attempts to modernize IEC 61131-3 by the IEC 61499 standard [3]. A third side argues that the proposed changes are not fundamental and are rather cosmetic in nature [4]. Additionally, it should be noted that possibility of reconfiguring and porting systems to distributed platforms of various topologies is initially declared in IEC 61499, and it is undoubtedly an essential attribute of the Industry 4.0 concept. IEC 61499 defines a program as a collection of interconnected and communicating function blocks. The external interface for the blocks is set up in data connections and event connections sections. A function block encapsulates the desired functionality which is specified by algorithms implemented in IEC 61131-3 languages. These algorithms are activated depending on the incoming events.

IEC 61499 therefore allows an automata-based description of system behavior. This is directly stated by the standard authors.

The effectiveness of automata-based approaches has long been recognized by the time IEC 61131-3 was submitted. Nevertheless, such mechanisms were not adopted until IEC 61499.

This evokes mixed thoughts. No doubt, it is a significant event for the mainstream practice of industrial automation. And yet it only happened decades after the adoption of IEC 61131-3. Besides, there is a perception that the standard does not fully take account of the common practices in automaton programming.

At implementation level, the behavior of function blocks is specified in the Execution Control Charts section with the Execution Control Chart (ECC) language. An ECC is roughly equivalent to a Moore-type state machine [5]. It monitors the input events and, based on the current state it executes a certain part of the encapsulated functionality. ECC is a graphical notation which leads to a possibility of ambiguous interpretation.

The order of ECC transitions' evaluation follows their order in textual XML-based representation of the FB. However, in graphical representation no hints provided to determine the order. This can result in two ECCs looking identically, but producing completely different reactions [5]. Also [4] lists such examples and argues that one of the most important reasons the industry has not yet adopted this standard is confusion over the execution semantics of the IEC61499 FB model.

The rest of the paper has the following structure: in Sect. 2, we give a basics of poST languages; in Sect. 3, we present a wheelchair lift as an example for control software specification; in Sect. 4, we describe control software structure; in Sect. 5, we illustrate using poST for FB specification; and in Sect. 6, we discuss the result.

2 Introduction to PoST

To address the restrictions and challenges in development of present-day complex control software, the process-oriented programming (POP) has been suggested in [6]. Process-oriented programming (POP) involves specifying control software with a set of concurrently running processes. Internally the processes have a state-machine-like structure and are equipped with operations for managing time intervals and inter-process communication. Concurrent behavior of the system is arranged via consequent execution of active process states on each program cycle. Compared to other known state-machine-based approaches, such as CSP [7], Input/Output Automata [8], Harels State-charts [9], Hybrid Automata [10], Esterel [11], Calculus of Communicating Systems [12], and their extensions [13,14], the POP approach combines system concurrency on the global scale with local linearity of behavior within each process. POP provides a conceptual basis for multiple domain-specific programming languages (DSLs) that are intended for natural control software specification.

Within this paradigm new C-like languages Reflex and IndustrialC [15,16] have been developed. The Reflex language targets PC-based control software for large-scale industrial applications, while IndustrialC is tailored for microcontroller-based embedded systems.

As practice shows, the Reflex language can be successfully used in industrial applications and offers a number of significant advantages in control software programming [17–19]. However, its widespread use in practice is hindered by the conservative nature of the domain. The developer community tends to be wary of introducing any new emerging technology to the process. Historically, the majority of control software is still implemented within the so called PLC-approach, that is based on the IEC 61131-3 languages IEC 61131-3 [1], and PLC manufacturers are reluctant to deviate from this standard.

In order to face this challenge we proposed to adapt the process-oriented approach for the ST procedural programming language in the same way as it was done for the C language in case of Reflex. The process-oriented extension of ST was called the poST language [20].

The poST language can therefore be of particular interest to the PLC community as it extends the Structured Text language from IEC 61131-3. The additional attractiveness of the poST language is due to the wide popularity of the ST language. According to the CoDeSys GmbH (former 3S-SmartSoftware Solutions GmbH), the ST language is regularly used by up to 70 % of users, and the number is constantly growing [21].

The poST language combines advantages of the process-oriented paradigm with conventional syntax of the ST language and can be easily adopted by the PLC community. The poST language assumes that a poST-program is a set of weakly connected concurrent processes, structurally and functionally corresponding to the technological description of the plant. Each process is specified by a set of states. The states are specified by a sequence of the ST constructs, extended by `TIMEOUT` operation, `SET STATE` operation, and `START`/`STOP`/check state operations to communicate with other processes. Apart from these operators, the poST language follows the syntax and semantics of ST.

For the poST language we have already developed an Eclipse-based IDE [22], including a parser and syntax-directed editor. Code generation modules for the

C and ST languages have been implemented. The generated ST-code can be automatically converted in the PLCopen XML Exchange format [23], which makes integration with the IEC 61131-3 tools easier. The approach assumes that the generated code will be translated to executable form and uploaded to the target platform with an existing C or IEC 61131-3 toolchain.

3 Wheelchair Lift Example

To demonstrate the approach, we choose the task of automating a lifting platform for low-mobility users (Fig. 1).

The lift is an alternative to a ramp and it is intended to overcome vertical barriers.

The *cyber-physical diagram* (Fig. 2) considers the system as three interacting components. The lift user acts as *Environment*. The lift acts as *Plant*. *Controller* defines the behavior of the system in accordance with the Reflex program. The user can press the external and internal call buttons as well as open and close the doors. The call buttons (*up_call*, *down_call*, *up_call*, *down_call*) and lift doors (*top_door_closed*, *bot_door_closed*) are used as *Controls*. LED indicators (*up_call_led*, *down_call_led*, *up_call_led*, *down_call_led*) are used to signal unhandled calls. The controller monitors the states of the controls and floor sensors (on_top_floor and on_bot_floor). Using the values of these inputs, the controller generates control signals (*up* and *down*) to the lift movement motor and controls the LED indicators. Turning on the motor causes the lifting platform to move between floors.

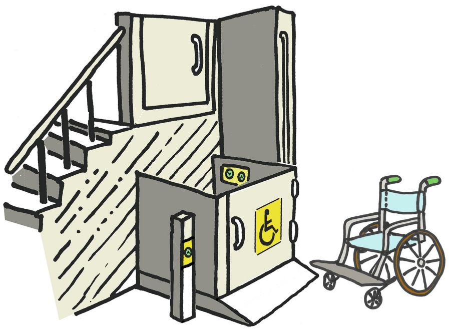

Fig. 1. Wheelchair lift for users with limited mobility

The control program includes the states of pressing each of the buttons, platform movement in both directions, and waiting for the end of a movement or a new command.

The requirements we use to design poST-program are formulated in natural language:

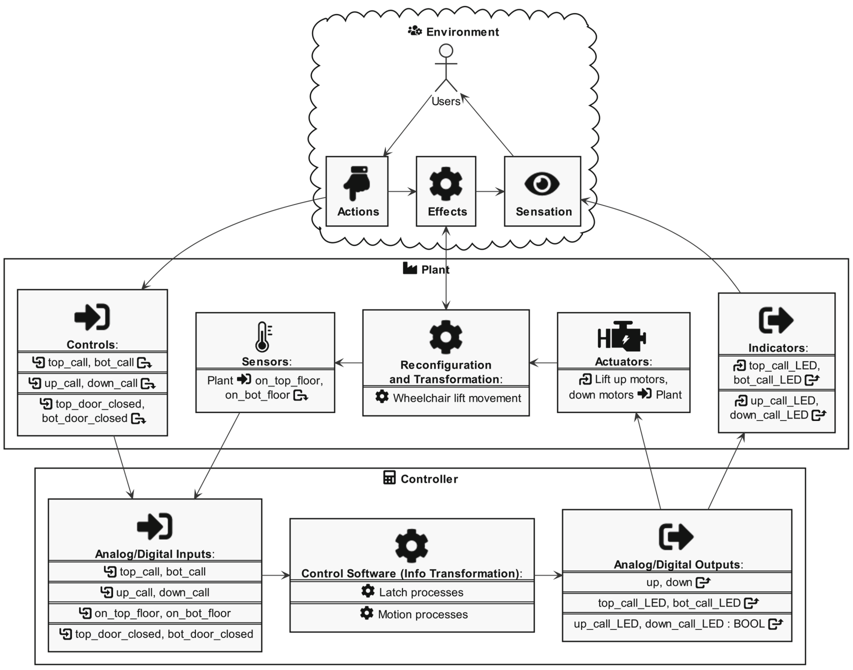
Environment
Users
Actions
Effects
Sensation
Plant
Controls:
top_call, bot_call
up_call, down_call
top_door_closed, bot_door_closed
Sensors:
Plant on_top_floor, on_bot_floor
Reconfiguration and Transformation:
Wheelchair lift movement
Actuators:
Lift up motors, down motors Plant
Indicators:
top_call_LED, bot_call_LED
up_call_LED, down_call_LED
Controller
Analog/Digital Inputs:
top_call, bot_call
up_call, down_call
on_top_floor, on_bot_floor
top_door_closed, bot_door_closed
Control Software (Info Transformation):
Latch processes
Motion processes
Analog/Digital Outputs:
up, down
top_call_LED, bot_call_LED
up_call_LED, down_call_LED : BOOL

Fig. 2. Wheelchair lift cyber-physical diagram

– The simultaneous appearance of *up* and *down* signals is prohibited.
– Platform movement is only possible with closed doors.
– The movement begins after pressing one of the call buttons.
– In the absence of control commands for 20 s, the lift should be automatically moved to the lower floor.

4 Software Design and Implementation in PoST

The process diagram (Fig. 3) depicts the poST-program structure. Initially, the *Initialization process* deploys the control algorithm. It starts the *Auxiliary processes* and the *Motion process*. Then it stops itself. The Auxiliary processes (the *xxx_call_Latch* processes) light up the LEDs when the user releases the corresponding call buttons and turns them off when the lift arrives at the corresponding floor. In accordance with the requests and the state of the sensors, the *Motion* process starts the *go_up* or *go_down* processes. After starting a process, the *Motion* process controls its completion (transition to the inactive state).

Each latch process has two local variables and two states. The local variables store previous values of the `top_call` and `top_door_closed` signals and are used to detect the rising and falling edges. In the initial state the process sets the starting values of its local variables. In its second and main state the process monitors the

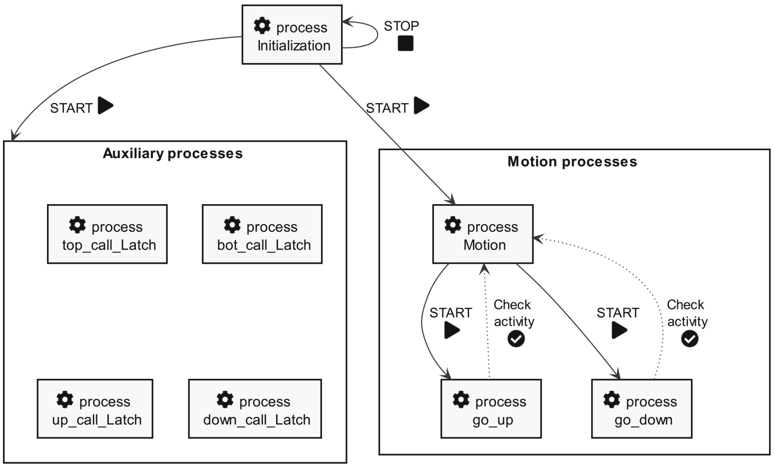

Fig. 3. Wheelchair lift process diagram

button press and door opening events via `top_call`and `top_door_closed` signals. On the button press event it sets the `top_call_LED` signal. On the door opening event the process turn off the LED. The latch processes (`top_call_Latch`, `bot_call_Latch`, `up_call_Latch`, `down_call_Latch`) differ only in variables (Listing 1).

```
PROCESS top_call_Latch
    VAR
        prev_in : BOOL;
        prev_out : BOOL;
    END_VAR
    STATE init
        prev_in := NOT top_call;
        prev_out := top_door_closed;
        SET NEXT;
    END_STATE
    STATE check_ON_OFF LOOPED
        IF top_call AND NOT prev_in THEN
            top_call_LED := TRUE;
        END_IF
        IF NOT top_door_closed AND prev_out THEN
            top_call_LED := FALSE;
        END_IF
        prev_in := call0;
        prev_out := open0;
    END_STATE
END_PROCESS
```

Listing 1. Latch process

The flowchart (Fig. 4) shows operation of the `Motion` process. Listing 2 represents the `Motion` process in poST.

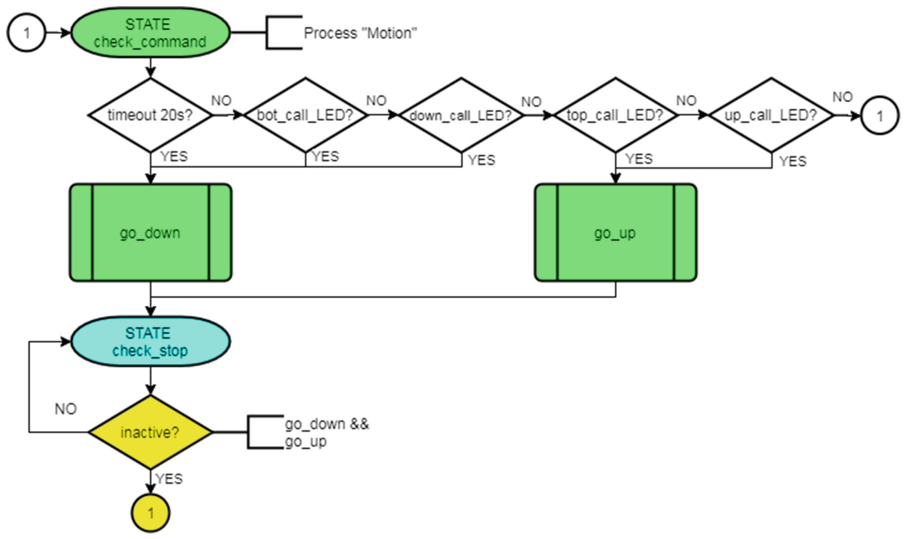

Fig. 4. Motion process flowchart

```
PROCESS Motion (* motion *)
    STATE check_command
        IF  (bot_call_LED) THEN
            START PROCESS go_down;
            SET STATE check_stop;
        ELSIF (down_call_LED) THEN
            START PROCESS go_down;
            SET STATE check_stop;
        ELSIF (top_call_LED) THEN
            START PROCESS go_up;
            SET STATE check_stop;
        ELSIF (up_call_LED) THEN
            START PROCESS go_up;
            SET STATE check_stop;
        END_IF
        TIMEOUT (#T20s) THEN
            START PROCESS go_down;
            SET STATE check_stop;
        END_TIMEOUT
    STATE check_stop
        IF ((PROCESS go_down IN STATE INACTIVE)
            AND (PROCESS go_up IN STATE INACTIVE)) THEN
            RESTART; // set the initial state
        END_IF
    END_STATE
END_PROCESS
```

Listing 2. Motion process

Code in Listing 3 defines the behavior of the `go_up` process. The process waits for both top and bottom doors to be closed, before setting for upward motion. The process stops once the lift reaches the top floor.

```
PROCESS go_up
    STATE motion
        IF (top_door_closed AND bot_door_closed) THEN
            up := TRUE;
        END_IF
        IF (on_top_floor) THEN
            up := FALSE;
            STOP;
        END_IF
    END_STATE
END_PROCESS
```

Listing 3. go_up process

5 Conceptual Implementation Within IEC 61499

Conceptually (Fig. 5), the algorithm above can be implemented with a single reduced function block with one Event Input invoking the algorithm specified in poST.

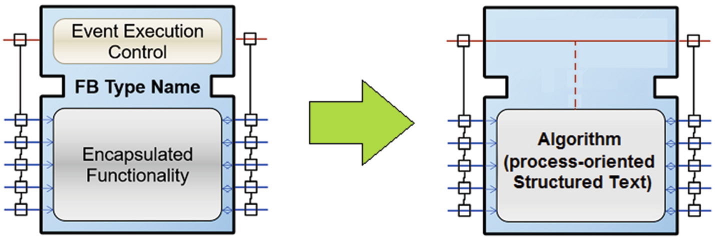
Event Execution Control
FB Type Name
Encapsulated Functionality
Algorithm (process-oriented Structured Text)

Fig. 5. Process-oriented function block

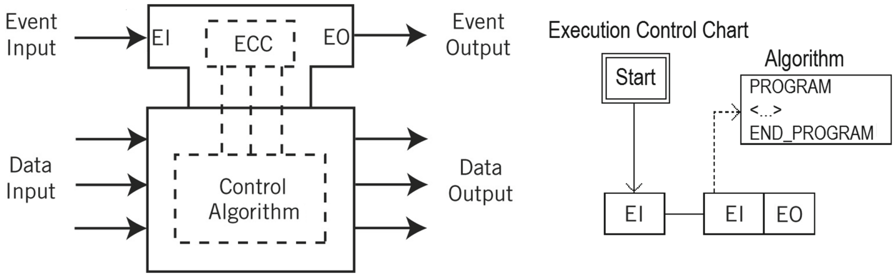

Fig. 6. ECC of process-oriented function block

Figure 6 depicts the internal structure of such a reduced function block.

This approach to implementation allows for the poST algorithm to be split into multiple function blocks. These can then be mapped unto multiple computing platforms thus enabling distributed control. Communication between the parts of the algorithm executing on separate PLCs can be organized using the IEC 61499 Virtual Bus concept, as seen on Fig. 7 and would primarily be presented by interaction between processes.

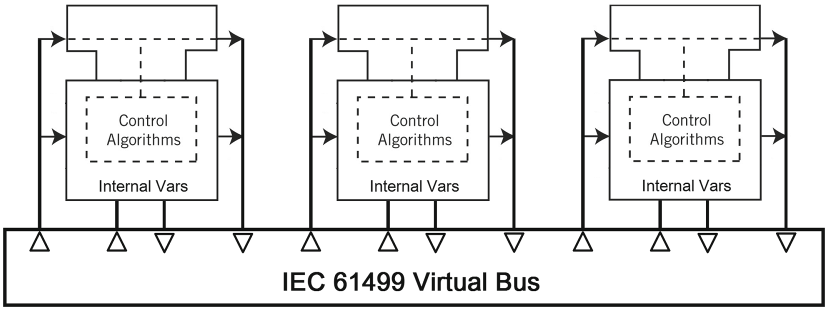

Fig. 7. Distributed implementation with IEC 61499 Virtual Bus

6 Discussion and Conclusion

The industrial systems market within the framework of the Industry 4.0 concept needs open solutions that can be implemented on distributed platforms. Currently, these are an open question for which a simple and effective answer needs to be worked out. IEC 61499 offers some solutions primarily attractive due to their ability for creating distributed control systems. However, this standard is highly dependent on the 30 year old standard IEC 61131-3. The 61131-3 standard is based on a device-centric paradigm and can only be successfully modified using existing process-oriented programming techniques. This shift to the application-centric paradigm can lead to a very steep learning curve. We propose to smooth the learning curve by using the poST language – a process-oriented extension of ST. Preliminary studies show that poST is compatible with the FB concept; moreover, when poST and the IEC 61499 are used, a synergistic effect appears. This synergy manifests in that specifying IEC 61499 function blocks in poST provides an application-centric paradigm for control software development with distributed architectures.

As a further direction of this research, we plan to formalize a new operational semantics of the poST language to align it with the IEC 61499 virtual bus features. An operational semantics would allow to adapt existing approaches in formal verification of process-oriented programs to the distributed case.

Acknowledgments. We thank the JetBrains Foundation for the charitable support of our research activity.

References

1. IEC 61131-3: Programmable Controllers Part 3: Programming Languages. International Electrotechnical Commission Std., Rev. 2.0 (2003)
2. Crater, K.C.: When Technology Standards Become Counterproductive. Control Technology Corporation (1992). https://support.controltechnologycorp.com/index.php?option=com_content&view=article&id=188&Itemid=null
3. IEC 61499: Function Blocks for Industrial Process Measurement and Control Systems, Parts 1–4. International Electrotechnical Commission Std., Rev. 1.0 (2004/2005)
4. Thramboulidis, K.: Different perspectives [face to face; "IEC 61499 function block model: facts and fallacies"]. IEEE Ind. Electron. Mag. **3**(4), 7–26 (2009). https://doi.org/10.1109/MIE.2009.934788
5. Vyatkin, V.: The IEC 61499 standard and its semantics. IEEE Ind. Electron. Mag. **3**(4), 40–48 (2009)
6. Zyubin, V.E.: Hyper-automaton: a model of control algorithms. In: Stukach, O. (ed.) Proceedings of the IEEE International Siberian Conference on Control and Communications, Tomsk, Russia, pp. 51–57. IEEE (2007). https://doi.org/10.1109/SIBCON.2007.371297
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall Inc. (1985)
8. Lynch, N., Tuttle, M.: An introduction to input/output automata. CWI Q. **2**, 219–246 (1989)
9. Harel, D.: Statecharts a visual formalism for complex systems. Sci. Comput. Program. **8**, 231–274 (1987)
10. Milner, R.: Communication and Concurrency. Series in Computer Science, Prentice Hall, Englewood Cliffs (1989)
11. Berry, G.: The foundations of Esterel. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner. Foundations of Computing Series, pp. 425–454. MIT Press (2000)
12. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: Timed I/O automata: a mathematical framework for modeling and analyzing real-time systems. In: Proceedings of the IEEE 24th International Real-Time Systems Symposium (RTSS 2003), Cancun, Mexico, pp. 166–177. IEEE Computer Society (2003)
13. Kof, L., Schätz, B.: Combining aspects of reactive systems. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 344–349. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-39866-0_34
14. Henzinger, T.A.: The theory of hybrid automata. In: Inan, M.K., Kurshan, R.P. (eds.) Verification of Digital and Hybrid Systems. NATO ASI Series (Series F: Computer and Systems Sciences), vol. 170, pp. 265–292. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-642-59615-5_13
15. Anureev, I., Garanina, N., Liakh, T., Rozov, A., Schulte, H., Zyubin, V.: Towards safe cyber-physical systems: the reflex language and its transformational semantics. In: International Siberian Conference on Control and Communications (SIBCON), Tomsk, Russia, pp. 1–6 (2019). https://doi.org/10.1109/SIBCON.2019.8729633
16. Rozov, A.S., Zyubin, V.E.: Adaptation of the process-oriented approach to the development of embedded microcontroller systems. Optoelectron. Instrum. Data Process. **55**(2), 198–204 (2019). https://doi.org/10.3103/S8756699019020122
17. Liakh, T.V., Rozov, A.S., Zyubin, V.E.: Reflex language: a practical notation for cyber-physical systems. Syst. Inform. (12), 85–104 (2018)

18. Kovadlo, P.G., et al.: Automation system for the large solar vacuum telescope. Optoelectron. Instrum. Data Process. **52**(2), 187–195 (2016). https://doi.org/10.3103/S8756699016020126
19. Liah, T.V., Zyubin, V.E.: The Reflex language usage to automate the large solar vacuum telescope. In: 2016 17th International Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices (EDM), Erlagol, Russia, pp. 137–139 (2016). https://doi.org/10.1109/EDM.2016.7538711
20. Bashev, V., Anureev, I., Zyubin, V.: The post language: process-oriented extension for IEC 61131-3 structured text. In: 2020 International Russian Automation Conference (RusAutoCon), Sochi, Russia, pp. 994–999 (2020). https://doi.org/10.1109/RusAutoCon49822.2020.9208049
21. Petrov, I., Wagner, R.: Debugging applied PLC software in CoDeSys (part 3). In: Industrial Automatic Control Systems and Controllers, vol. 4, pp. 34–36. Nauchtekhlitizdat (2006)
22. Wiegand, J.: Eclipse: a platform for integrating development tools. IBM Syst. J. **43**(2), 371–383 (2004). https://doi.org/10.1147/sj.432.0371
23. Marcos, M., Estevez, E., Perez, F., Der Wal, E.V.: XML exchange of control programs. IEEE Ind. Electron. Mag. **3**(4), 32–35 (2009). https://doi.org/10.1109/MIE.2009.934794

Author Queries

Chapter 17

Query Refs.	**Details Required**	**Author's response**
AQ1	This is to inform you that corresponding author has been identified as per the information available in the Copyright form.	