IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 6 February 2022, accepted 22 February 2022, date of publication 8 March 2022, date of current version 6 April 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3157601

poST: A Process-Oriented Extension of the IEC 61131-3 Structured Text Language

VLADIMIR E. ZYUBIN[1], ANDREI S. ROZOV[1], IGOR S. ANUREEV[1], NATALIA O. GARANINA[1], AND VALERIY VYATKIN[2,3], (Fellow, IEEE)

[1]Institute of Automation and Electrometry SB RAS, 630090 Novosibirsk, Russia
[2]Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, 97187 Luleå, Sweden
[3]Department of Electrical Engineering and Automation, Aalto University, 02150 Espoo, Finland

Corresponding author: Valeriy Vyatkin (valeriy.vyatkin@aalto.fi)

This work was supported by the Russian Ministry of Education and Science under Project AAAA-A19-119120290056-0, and in part by the H2020 Project 1-SWARM co-funded by the European Commission under Grant 871743.

ABSTRACT This paper presents the core concepts for the poST language – a process-oriented extension of the IEC 61131-3 Structured Text (ST) language which intends to provide a conceptual consistency of the PLC source code with technological description of the plant operating procedure. The poST can be seamlessly used as a textual programming language for complex PLC software in the context of IEC 61131-3 (3^{rd} Edition). The language combines the advantages of FSM-based programming with the conventional syntax of the ST language which would facilitate its adoption. 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 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 basic syntax of the poST language, demonstrates the usage of the poST language by developing control software for an elevator, and compares the development in poST with pure Structured Text.

INDEX TERMS Computer languages, IEC 61131-3 standard, PLC software development, process-oriented programming.

I. INTRODUCTION AND MOTIVATION

This work is motivated by the fact that software engineering for industrial automation systems based on the IEC 61131-3 set of languages [1] gets increasingly complex while the software development methods, in general, are evolving toward concepts that offer a higher level of abstraction.

The IEC 61131-3 standard consists of five languages: (1) the single register assembly language Instruction List (IL), (2) The Pascal-like Structured Text language (ST), (3) Ladder Diagram (LD), based on a metaphor relays used for automation in the pre-digital era, (4) the Function Block Diagram (FBD) language, implementing dataflow programming for continuous control, (5) the Sequential Function Chart (SFC) language based on the Petri nets model.

Overall, the standard is consistent with the specific features of industrial control. Industrial controllers are inherently open (i. e. communicate with an external environment), reactive (have event-driven behaviour) and concurrent (have to process multiple asynchronous events). These features call for special languages to be used in development of control software, e.g. the IEC 61131-3 languages [1] which are the most popular in the PLC domain. However, the technology behind IEC 61131-3 imposes several restrictions for the development of reactive systems and especially of today's complex and safety-critical systems [2]–[4]. This motivates researchers to enrich the IEC 61131-3 development model [5]–[8], or develop alternative approaches, e. g. [9]–[13], [15].

A significant number of researchers believe that such a development can be provided through the adaptation of the Finite State Machine (FSM) concept by the standard. This conclusion is supported by attempts to extend the IEC languages with finite automata concepts, e.g., [16], [17]. Furthermore, automata-based patterns are widely used in PLC programming [18]–[25], and the domain of reactive system programming (including PLC) has seen a long history of utilizing and further developing automata-based concepts.

The associate editor coordinating the review of this manuscript and approving it for publication was Roberto Nardone.

While the SFC aims at applications where sequential nature of processes imposes the corresponding program structure, this is a graphical program representation. Although diagrammatic visual models and languages are very popular in industrial automation, nevertheless, there is a great number of software developers who prefer textual programming languages, for which reason the ST language has a very significant share among the developers in many countries. However, its cyclic-scan-based PLC legacy does not allow for an intuitive implementation of the state-machine-based logic, requiring using such constructs as IF-THEN-ELSE and CASE for implementation of the state transition function. This complicates the code and makes its static analysis harder. In this paper we propose a solution in the form of the poST programming language which has dedicated higher-level constructs for describing the state-based logic of sequential processes.

The structure of this paper follows a classical pattern for introducing novel formal languages as in e.g. [26], [27], using samples of code in the proposed poST language to illustrate its properties. In the following section, the related concepts and works are analyzed. The main contribution is presented in four parts. First, we describe the concept of process-oriented programming and formulate the hypothesis of the research (Section III). The second part contains the basic syntax of the poST language (Section IV). In part three we depict an Eclipse-based poST IDE (Section V). And finally, we empirically demonstrate the poST features using an example of a three-floor elevator controller (Section VI). In the concluding section, we discuss the paper's contribution and outline possible development paths.

II. RELATED WORKS

A. STATE MACHINE CONCEPTS FOR REACTIVE SYSTEMS

The use of the FSM-based programming paradigm in control dates back at least to the late 1970s when the early implementations of Quickstep were developed [28]. In the beginning, it was used without a strong mathematical foundation, which was not perceived as necessary.

In [26] Harel introduced Statecharts notation for specification of complex reactive systems. Statecharts is based on the hierarchical state machine (HSM) model and presents a control system as a set of concurrently executing and interacting nested automata.

To model the behavior of asynchronous reactive systems the notion of timed Buchi automata (TBA) was proposed [29], which is Buchi automata where events are coupled with non-negative real numbers (time values). In [30] it has been shown that TBA can be used for modeling and analyzing the timing behavior of reactive software. The methods for checking reactive software properties have been implemented in a variety of tools, including the model checkers UPPAAL [31] and Kronos [32]. However, despite 20 years of research, these developments have not widely reached industrial attention.

Methods based on the FSM model are actively used in formal verification of digital systems using model-checking (e.g. SPIN [33], COSPAN [34], SMV [35]). Of these languages, FSMs are most extensively used in SMV, which provides case/switch constructs for low-level description of finite automata [36]. It also provides the ability to create networks of communicating state machines. The interaction between automata is organized through input-output variables (signals) in a data-flow style [37].

Dierks in [18] introduces PLC-automata – an automata-based formalism for simulation and static time analysis of reactive systems. PLC-automaton enriches the classic finite-state automaton with an upper bound for the cycle and assigns to each state a set of delayed inputs and a delay time for how long the delayed inputs should be ignored. The authors specify a formal semantics for the PLC-automata and suggest methods for translation into ST code. The formal semantics make the formalism suitable for verification using model checkers. PLC-automaton is presented as a language for PLC systems specification, however, due to loose syntax definition, the formalism seems hardly usable for practical PLC programming. Though it can be used as a basis for an actual programming language, such a language is not described in the work.

Kaynar *et al.* in [38] define timed I/O automata. This model combines typed variables, a system state composed of variable values, external and internal actions, discrete transitions, and trajectories. The model has enhanced expressiveness compared to the model of Alur-Dill automata [29]. This work reflects the demand for an automaton model equipped with such concepts as variables, events, and time.

Sacha in [39] presents a finite state time machine (FSTM) formalism that enhances the classic Moore automaton with timer symbols and a timer function. This extension enables defining state transitions depending, apart from the input symbols, on the value of a timer, which is started upon entering the state. Unlike in a Moore automaton, the response of a finite state time machine depends on the time intervals within the sequence of input symbols. The authors demonstrate a development process using this model. The automaton is specified using UML state diagrams which can then be automatically translated into PLC-executable code in ladder diagram notation.

Zyubin in [40] proposes the hyper-automaton model for control algorithms. In this model, the classical FSM is enhanced with inactive states, a time-in-state timer. This enhanced FSM is called a process. The control system is expressed with multiple concurrent processes. A process state is defined by a set of events and reactions to the events. An event denotes a superposition of changes in the process states, time-in-state timer, and system input values. A reaction is defined as a superposition of operations changing the process states, time-in-state timer, and system output values. Unlike the Harel statecharts, the processes have no nested states, yet system hierarchy can still be expressed via inter-process communication. Execution semantics of the hyper-automation vary depending on the concrete implementation. In the simplest case, the processes

execute consequently and cyclically in a cooperative concurrency. In [41] Rozov and Zyubin presented an extension of the model with multiple hyper-automata executing with interrupt-driven preemption.

Encapsulation of state machines into IEC 61499 function blocks for implementing both controllers and models of the plant has been demonstrated in [42].

B. STATE MACHINE PROGRAMMING IN INDUSTRIAL APPLICATIONS

Katzke and Vogel-Heuser in [6] propose the integration of concepts like statecharts into classical IEC 61131-3 environments. This tool supports 3 types of UML diagrams, two of which, Statecharts and Activity Diagrams, were made fully executable up to the level of PLC programming languages. A variant of the statechart implemented in the MATLAB framework is used to model reactive systems within a Simulink model [43].

Secchi, *et al.* [44] have introduced the use of UML in automation providing methodology of PLC code design by refining UML specification. Thramboulidis proposed in [45] generation of IEC 61499 function blocks from UML diagrams. In [5] they further compare the IEC 61499, UML, SysML approaches to facilitate model-driven in IEC 61131-3 and conclude that SysML can be used to model control software and automatically generate IEC 61131 code. Similar attempts were made for the IEC 61499 standard [46]. Basic function blocks of IEC 61499 are programmed using a state machine called Execution Control Chart (ECC). Examples of system designs with state machines can be found in [47]–[49].

Frey in [50] suggests using Signal Interpreted Petri Nets (SIPN) that have enhanced dynamics as compared to SFC. In SIPN several transitions can fire simultaneously and there can be iterated firing of transitions in between stable markings.

In the TinyOS/nesC project [51] authors proposed the nesC language [52] oriented for microcontroller-based reactive systems. Based on the results of language application in embedded systems practice, developers discovered many code segments organized in the form of finite state machines (FSMs). The absence of embedded tools for code organization in the FSM form was noted as a drawback of the approach in [53].

Samek in [12] classifies standard state machine implementation techniques in high-level programming languages, such as C or C++. According to the author, the main techniques are the nested switch statement, the state table, and the object-oriented State design pattern. Shalyto *et al.* [13], [14] promote structuring control programs as state machines in the C language using the switch statement. Wagner in [11] proposed to represent a reactive system as a set of concurrently operating state machines.

Zyubin in [15] proposes the SPARM language – a C-like language that describes a control algorithm as a set of weakly connected concurrent processes. A SPARM-process is an FSM-like entity (a set of process states) where C-operators and sentences are enhanced with inter-process operators and PAUSE-states to provide concurrent and timed behavior of the software. The language was implemented for the time-triggered polling-based model of execution. The language was successfully used in the process control system for a silicon single-crystal growth furnace [54] that practically proves acceptance of the approach for complex control algorithms.

This idea saw further development in the hyperprocess model [40] employed in the Reflex language [55]. The model was, in part, inspired by [56] and features built-in time-out instructions. The hyperprocess was later adopted in the IndustrialC language [41], [57] aimed at the development of microcontroller software and embedded systems.

These languages are C-like and therefore they are easy to learn. Translators of the languages produce C-code and therefore cross-platform portability is achieved. With their native support for state machines and floating-point operations, these languages allow cyber-physical systems to be easily expressed. Unfortunately, the IEC 61131-3 languages do not use C-like notation, and an attempt to include a new C-like language in the IEC 61131-3 set would fail even if the language is an extremely powerful one.

However, the IEC 61131-3 standard includes a Pascal-like language Structured Text (ST). ST is a high-level language and has ample expressive power as compared to other low-level languages of the standard, e.g. Ladder Diagrams [58]. The attractiveness of the language is also due to its wide popularity. According to CoDeSys GmbH (former 3S-SmartSoftware Solutions GmbH) ST is regularly used by up to 70% of users, and the number is constantly growing [59]. Moreover, a noticeable part of ST developers use the FSM pattern for their programs.

We therefore can adapt the process-oriented approach for this procedural programming language in the same way as it was done for the C language. To do this, we develop a process-oriented extension of the ST, or in abbreviated form, poST.

III. THEORETICAL FRAMEWORK FOR THE PROPOSED EXTENSION

In this section we present the fundamental mathematical and terminological grounds for our proposed programming language.

A. CONTROL SOFTWARE FEATURES

When considering a modern control system, we generally picture a digital controller connected to a controlled plant. The plant represents the external environment of the control system and consists of hardware and equipment within which physical processes take place. The plant and controller are connected via sensors and actuators. Sensors read data from the plant and pass it to the control system. The controller then acts or rather reacts on this input by producing control values for the actuators, which in turn alter the flow of the physical processes in the plant.

Input from the sensors is supplied continuously. The controller detects events within the data flow and consequently reacts to them as determined by the control program. Therefore, unlike transformational software, control algorithms operate indefinitely, which for digital controllers automatically implies cyclic execution. A general control algorithm is thus presented with the following pattern: input of data about the current state of controlled plant from sensors – data processing and determination of required reaction – and data output, which changes the current state of the actuators and consequently the state of the plant.

Technological processes tend to involve multiple stages and the way control software reacts to events needs to change over time. Control algorithms have polymorphic behavior which cannot be defined by the knowledge of inputs alone but depends on a history of events.

As the plants are dynamic systems, control software has to function time-dependently, i. e. according to the plant dynamics. This means that control algorithms accommodate delays, latencies, idles, pauses, watchdogs, timeouts, and other notions connected to time intervals.

Another important feature of control algorithms is that they are almost always highly concurrent. The system needs to control multiple physical processes and communicate with a variety of hardware peripherals – all at the same time. As physical processes in the plant evolve independently, the sequence of events is arbitrary. Therefore any attempts to describe the control system within a single monolithic block lead to a combinatorial explosion of complexity [60]. Avoiding this requires the system to be split into a multitude of independent or loosely coupled control flows.

Lastly, we ought to mark the hierarchical structure of any complex control algorithm that reflects the artificial nature of the external environment, the designer plan that is implemented in the architecture of the facilities [61]. Taking into account the previous remarks we can say that the hierarchical structure consists of chains that are independently executed in parallel. This means that divergence and convergence of control flow are the base for a significant part of the control algorithm. The algorithm structure can be arbitrary, irregular, and looped.

To summarize our analysis, we list the key features of digital control systems we deem most significant:

- interaction with an external environment via sensors, actuators, controls, and indicators,
- indefinite running time,
- cyclic execution,
- event-driven polymorphic behavior,
- operations with time intervals,
- concurrency,
- hierarchical structure.

In this paper we propose a programming language PoST based on the extended model of finite state machines and hyperprocesses. We hypothesize that this model and the PoST language provide tangible benefits to the developers in the domain of industrial automation systems. The hypothesis is validated in a constructive way: in the following subsection, the hyperprocess model is introduced. Then, the syntax and semantics of PoST are presented in detail, followed by a case study, demonstrating the use of the language in a concrete application scenario. The benefits of the proposed approach are discussed in Section VI.

B. HYPERPROCESS MODEL

From practice, it is apparent that the Finite State Machine (FSM) concept is particularly promising for use in logical control. FSMs implicitly assume the presence of an external environment, can execute cyclically, and exhibit event-driven polymorphic behavior. However, our analysis of control system features shows that the model also needs to support concurrency, hierarchy, and timed operations.

Furthermore, the FSM model is tailored for hardware implementation. This is due to historical circumstances of when the model was created [62]. Negative effects of this become apparent even on the conceptual level, in the terms ''input alphabet'' and ''output alphabet'' which, while simple and convenient for theoretical studies, appear awkward and obscure from a modern programmer's standpoint. This leads to general misunderstanding and discourages usage of this potentially very powerful concept [63]. Efficiency in practice requires that more conventional programming concepts, like variables and statements, be supported. With this in mind, we have constructed an FSM-based model of control software, that is suitable for the domain of control software development. Here we will outline its basic structure and key properties that are necessary to lay a foundation for poST language. A more detailed description of the hyperprocess model can be found in [40].

The control software is represented as a hyperprocess – an ordered set of processes, which are cyclically activated with the period T_H. Mathematically, the hyperprocess is defined by a triplet:

$$H =< T_H, P, p_1 >, \text{ where} \tag{1}$$

- T_H is the period of activation,
- P is a finite nonempty ordered set of processes ($P = p_1, p_2, \ldots, p_M$, where M is the number of processes),
- p_1 is the first marked process, $p_1 \in P$.

At this point, we can assume that a process is just a function or a set of instructions (in a programming sense). Note that the word ''ordered'' refers to the textual description of a program. It should also be noted that a kind of perfect synchrony hypothesis [64] is assumed in this model. In contrast to the original hypothesis, which states that neither computation nor communication takes any physical time, we assume a relaxed statement to be true: the latency period for calculation overhead is less than or equal to the period of activation T_H. This seems to be a less idealistic and quite reasonable condition for software implementation.

A process denotes a polymorphic subroutine – it is a set of mutually exclusive subroutines (i.e. blocks of sequential

program code) which is handled as a unified entity. We will further refer to these subroutines as process state functions. For any cycle of hyperprocess activation, each process is reduced to one of its state functions as determined by the current state of that process. That state function is called the current function of the process and provides instructions to be executed during that hyperprocess activation. In particular, it can contain instructions that change the state of the process for the next cycle. State functions containing no instructions are referred to as passive and correspond to inactive states of the process. Each process also keeps track of how many hyperprocess cycles it has spent in its current state.

Formally, i-th process is a quintuple

$$p_i =< F_i, F_i^p, f_i^1, f_i^{cur}, T_i >, \text{ where} \tag{2}$$

- $p_i \in H, i = 1, 2, \ldots, M$,
- F_i is a set of mutually exclusive functions,
- F_i^p is a set of mutually exclusive passive functions, $F_i^p \subset F_i$,
- f_i^1 is the first function (or marked active function), $(f_i^1 \in F_i) \wedge (f_i^1 \notin F_i^p)$,
- f_i^{cur} is the current function, $f_i^{cur} \in F_i$,
- T_i is the current time.

In programming, particularly in C, the term function is equivalent to a subroutine – a set of instructions or statements, that specify mathematical calculations, conditional actions etc. In the hyperprocess model we rather prefer to accentuate the event-driven and reactive features of the model. A state function is therefore defined as a set of events and reactions to the events. Formally, j-th functions of i-th process is a twain

$$f_{ji} =< X_{ji}, Y_{ji} >, \text{ where} \tag{3}$$

- X_{ji} is a set of events,
- Y_{ji} is a set of reactions.

As events, we consider any changes or superpositions of changes inside or outside the hyperprocess that are of importance to the control algorithm. The event is connected to a reaction it stimulates. Reactions are superpositions of actions, including calculations, change of output values, state transitions, communication with other processes, etc. A state with no events has has no reactions:

$$(X_{ji} = \varnothing) \Rightarrow (Y_{ji} = \varnothing). \tag{4}$$

Passive functions can then be defined as follows in terms of events and reactions:

$$(f_{ji} \in F_i^p) \Leftrightarrow (X_{ji} = \varnothing). \tag{5}$$

In essence, the process concept is a modification of the FSM model. The input and output alphabets have been removed and states of automaton along with its transition relation have been replaced with state functions. Transitions between states are part of the instructions within state functions. The input and output alphabets have been replaced with events and reactions. The transition relation of automaton is spread across state functions and expressed with a special reaction:

$$set\_state(p_i, f_i^j) \equiv (f_i^{cur} := f_i^j, T_i := 0).$$

Thus, the original FSM model was preserved within the process model. Describing software with multiple automata provides concurrent execution with a granularity of the functions. The hyperprocess execution can be described in the following way: the hyperprocess is cyclically activated with a period T_H. upon each activation the current state function f_i^{cur} for each process $p_i \in P$ is executed. With each activation, the time T_i for the process is incremented. Execution of a state function consists of sequentially testing for each of its monitored events and executing corresponding reactions for events that are detected.

To provide means for time-tracking and communication between processes, special events and reactions are defined. A process can check whether another process is in a passive state:

$$passive(p_i) \Leftrightarrow (f_i^{cur} \in F_i^p).$$

For time tracking purposes, a timeout event can be monitored:

$$timeout(p_i, T_{timeout}) \Leftrightarrow (T_i > T_{timeout}).$$

This event is triggered once the process p_i has been executing with the same state function for a number of hyperprocess cycles given by $T_{timeout}$. To allow divergence and convergence of control flow, processes can start and stop other processes:

$$start(p_i) \equiv (f_i^{cur} := f_i^1, T_i := 0),$$
$$stop(p_i) \equiv (f_i^{cur} := f_i^{stop}), \text{ where } f_i^{stop} \in F_i^p.$$

These two reactions in conjunction with the *passive* event facilitate the arrangement of the processes into a hierarchic structure, with higher-level processes starting lower-level processes being a rough equivalence of calling subroutines in procedural programming. With this model, we, therefore, have preserved the original cyclic, event-driven, and polymorphic nature of the FSM model, and extended it to support concurrency, hierarchy, and time tracking.

IV. POST LANGUAGE SYNTAX AND SEMANTICS

The hyperprocess model presented above underlies the poST language, which enriches the ST language. The language uses ST syntax for the statements, expressions, and variable declarations. Additional constructs are implemented, for process definitions, process interaction, and tracking of time intervals. Within standard terminology, a poST program is an IEC 61131-3 program, which, according to the POU concept, can be executed once, on a timer, or upon an event. The IEC 61131-3 runtime environment manages cyclic execution of the poST program via time-triggered calls of the program code.

A poST program consists of process definitions. A process is defined by its local variables and a list of its states. The first defined state in the list is the process's initial state.

States are lists of ST statements, extended by TIMEOUT, SET STATE operations, as well as START / STOP / check state operations to communicate with other processes. Each process also has two implicit inactive states: the STOP state that is for a normal halt of the process, and the ERROR state indicating an abnormal halt. A process is inactive if it is in either of these two inactive states. Otherwise, the process is considered active. For a poST program, the initial process is one that is defined first. For a process, its initial state is one that is defined first. A sample process definition in the poST language is presented in Listing 1.

```
(* simple process *)
PROCESS simple_process
  STATE first_state
    out := TRUE;
    SET STATE second_state;
  END_STATE
  STATE second_state
    IF (inp = TRUE) THEN
      out := FALSE;
      STOP;
    END_IF
  END_STATE
END_PROCESS
```

Listing 1. Simple process expressed in the poST language.

Once started, the process sets the out to TRUE and changes its current state to second_state. It then repeatedly checks the value of inp on each consequent time-triggered call. If the variable reads TRUE, the process sets out to FALSE and stops. In the case process changes its current state to the next defined state, the statement SET NEXT is used for the short hand. The PoST notation allows for an intuitive implementation of an arbitrary state machine logic inside the process but is optimized for the most typical case of the sequential organization of states. A shorthand SET NEXT can be used for transition to the next defined state.

As in the hyperprocess model, on the first call, the initial process is in its initial state, while the rest of the processes are in the STOP state. An example of further unfolding control algorithms with interprocess operations is presented in Listing 2. Once started, the initial process starts the P_1 process and stops. The P_1 process implements the algorithm from the previously discussed code sample.

Apart from starting and stopping other processes, a process can monitor their status using ACTIVE/INACTIVE predicates. We demonstrate this kind of process interaction in Listing 3. Here the Init process starts P_1 and proceeds to state wait_1, where it monitors the P_1 status with an IN STATEACTIVE predicate. Once P_1 becomes inactive, the Initprocess starts P_2 and goes on to checking its state in wait_2. As soon as P_2 becomes inactive, the Init process stops.

Combined with the START statements, these status monitoring predicates provide a mechanism of program organization that is somewhat similar to subroutine calls in procedural languages. At the same time, starting a process means a divergence in control flow as, in effect, a new concurrent (logically independent) thread is created. Listing 4 demonstrates a

```
PROGRAM sample_program
  PROCESS Init (* the initial process *)
    STATE begin
      START PROCESS P_1;
      STOP;
    END_STATE
  END_PROCESS
  PROCESS P_1
    STATE first_state
      out := TRUE;
      SET NEXT; (* shorthand *)
    END_STATE
    STATE second_state
      IF (inp = TRUE) THEN
        out := FALSE;
        STOP;
      END_IF
    END_STATE
  END_PROCESS
END_PROGRAM
```

Listing 2. Initial unfolding of algorithm.

```
PROGRAM sample_program
  PROCESS Init (* the initial process *)
    STATE begin
      START PROCESS P_1;
      SET NEXT;
    END_STATE
    STATE wait_1
      IF (PROCESS P_1 IN STATE INACTIVE) THEN
        START PROCESS P_2;
        SET NEXT;
      END_IF
    END_STATE
    STATE wait_2
      IF (PROCESS P_2 IN STATE INACTIVE) THEN
        STOP;
      END_IF
    END_STATE
  END_PROCESS
<...>
END_PROGRAM
```

Listing 3. Process completion monitoring.

```
STATE diverge
  START PROCESS P_1;
  START PROCESS P_2;
  <...>
  START PROCESS P_N;
  SET NEXT;
END_STATE
```

Listing 4. Simultaneous starting multiple processes.

```
STATE converge
  IF ((PROCESS P_1 IN STATE INACTIVE) AND
      (PROCESS P_2 IN STATE INACTIVE) AND
        <...>
      (PROCESS P_N IN STATE INACTIVE)) THEN
        SET NEXT;
  END_IF
END_STATE
```

Listing 5. Control flow convergence.

control flow divergence with starting multiple processes simultaneously. Listing 5 demonstrates a control flow convergence with status monitoring predicates of multiple processes.

Combining divergence and convergence mechanisms, one can arrange the processes into an architecture with arbitrary parent-child relationships. However, in practice control software tends to be mostly hierarchical in structure.

Another poST construct extending the functionality of ST is the TIMEOUT statement. The statement consists of a time interval and a set of actions to be executed in the event

of a timeout. By definition, every process has a timer to generate the event. The timer is automatically reset upon state transitions. Alternatively, one can manually reset the timer using the `RESET TIMER` statement. Listing 6 demonstrates using the TIMEOUT statement to provide a 3-second delay before continuing further.

```
STATE delay3s
  TIMEOUT T#3s THEN
    out := FALSE;
    SET NEXT;
  END_TIMEOUT
END_STATE
```

Listing 6. Using TIMEOUT statement for delay.

The `TIMEOUT` statement is very useful for detecting and handling abnormal conditions of the plant, e.g. to check threshold time for closing a valve. Listing 7 shows the technique to close a shut-off valve.

```
PROCESS close_Valve
  STATE begin
    Valve_Control := OFF;
    SET NEXT;
  END_STATE
  STATE wait_closing
    IF (Valve_Sensor = OFF) THEN
      STOP;
    END_IF
    TIMEOUT T#1s THEN
      ERROR;
    END_TIMEOUT
  END_STATE
END_PROCESS
```

Listing 7. Using TIMEOUT to control abnormal situations.

Here the process initiates valve closing by setting the control signal to OFF and then monitors the sensor to ensure that the closing is finished. The timeout statement is used to detect abnormal situations that lead to the valve being stuck.

To facilitate studying the poST language, a poST-to-ST compiler has been developed in form of a web application and deployed at the following address: http://post2st.iae.nsk.su.

V. ELEVATOR CASE STUDY

We demonstrate programming in the poST language with an example of developing control software for a three-floor elevator.

A. THREE-FLOORS ELEVATOR DESCRIPTION

For our purpose, we use a simplified case of an elevator as given in [65]. Elevators themselves are simple devices, and basic lifting systems have not changed much in over 50 years. Control systems, however, have changed substantially to improve safety and speed of operation.

We consider an elevator servicing three floors. A single elevator motor and three separate door motors control the elevator position and door operation respectively. The elevator motor is activated by the `up` and `down` signals and moves the elevator between floors. Three position sensors are used to detect if the elevator is properly positioned at one of the three floors. Elevator doors have two panels that meet in the middle and slide open laterally. The electric door opening motors open both the inner elevator door and the outer floor door, and are activated by the open signals. When the elevator door opens, it is sensed by one of the three door limit switches (three door-opened signals). The elevator has three call buttons, one on each floor, to request the elevator from the floors (three call signals) and three inner buttons to set requested floors from the cabin (three button signals). Every button has a LED indicator to show the active requests.

The elevator operates under the following rules:

1) The elevator must continue traveling in the same direction as long as there are remaining requests in this direction.
2) If there are no further requests, the elevator must stop and become idle. If only requests for the opposite direction are present, the elevator must switch the preferred direction and start serving the requests.
3) The doors must always be closed when the elevator is moving.
4) When the cab has reached a proper location at a requested floor, the elevator must open the door and then close it after a three seconds pause.

B. CONTROLLER AND PLANT SIMULATOR DESIGN

The cyber-physical diagram in Fig. 1 represents the system as three interacting components: *Environment*, *Plant*, and *Controller*. In the discussed case study the elevator users act as an *Environment* with respect to the *Plant* and *Controller* which are connected with each other in closed-loop. The *Controller* defines the behavior of the system in accordance with the poST program. The user can press the external and internal call buttons. The floor call buttons (`call0`, `call1`, `call2`), the cabin call buttons (`button0`, `button1`, `button2`) are used as *Controls*. LED indicators (`call0_LED`, `call1_LED`, `call2_LED`) combined with the cabin call buttons, and LEDS (`button0_LED`, `button1_LED`, `button2_LED`) combined with the floor call buttons are used to signal unhandled calls. The floor LEDS (`floor0_LED`, `floor1_LED`, `floor2_LED`) indicates position of the elevator cabin.

The controller monitors the states of the controls, floor sensors (`on_floor0`, `on_floor1`, `on_floor2`) and door sensors (`door0closed`, `door1closed`, `door2closed`).

Using the values of these inputs, the controller generates signals for the elevator movement motor (`up` and `down`), door motors (`open0`, `open1`, `open2`), and the LED indicators. Turning on the elevator motor causes the elevator to move between floors, whereas the down signal overrides the up signal. Turning on a door motor causes that door to open while turning it off closes the door.

Our design, as presented on the diagram, assumes separate implementation of the controller, the plant simulator, and the effects of visualization. This approach allows debugging the code on a development platform while providing a seamless transfer of the verified program to the target PLC.

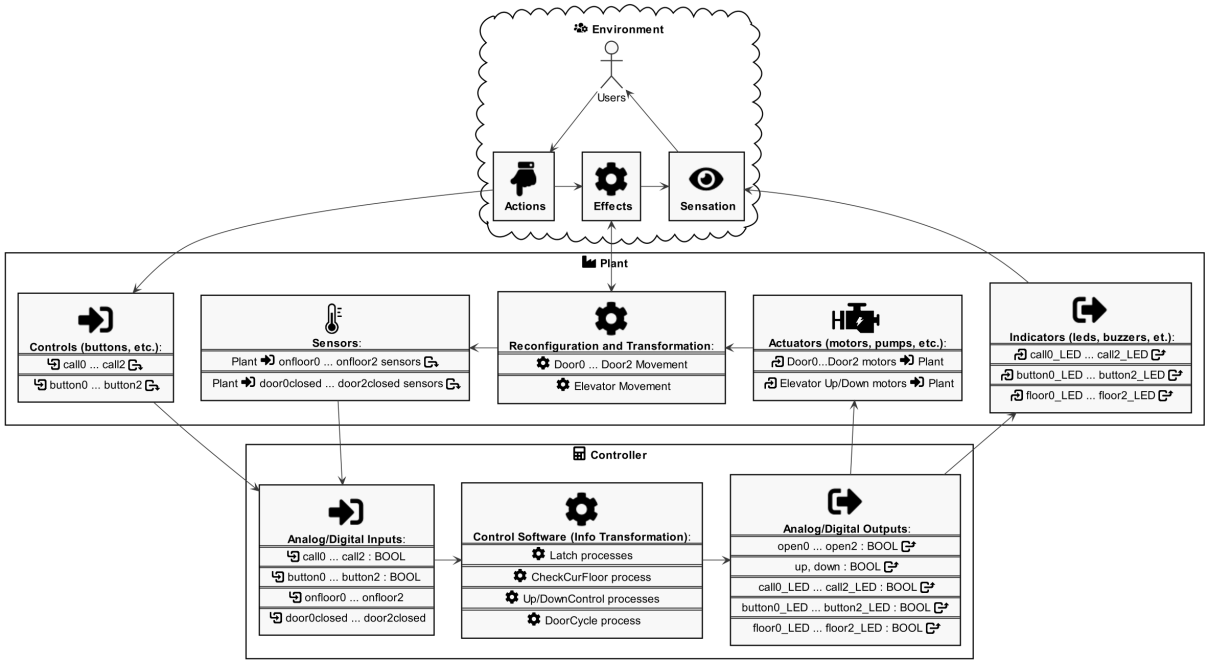


FIGURE 1. Elevator cyber-physical diagram.

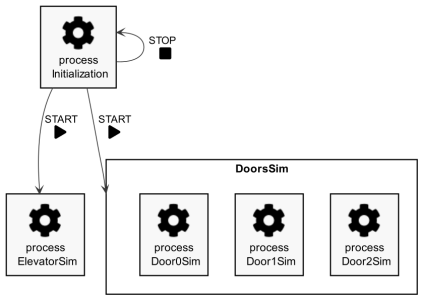


FIGURE 2. Plant simulator process diagram.

C. PLANT SIMULATOR IMPLEMENTATION IN POST

The architecture of the plant simulator program (Fig. 2) consists of five processes.

The initial process (Initialization process) launches processes that simulate moving the elevator between floors (the ElevatorSim process) and opening doors (Door0Sim …Door2Sim processes). After the launching, the process stops itself (Listing 8).

The simulation processes are similar one to another. For example, the Door0Sim process (Fig. 3.) consists of one active state. According to the `open0` signal, it calculates changes in the doors coordinate and limits it to the allowed values reflecting the size of the doorway. If the coordinate is equal to zero it imitates the corresponding door sensor by setting the `door0closed` signal to TRUE (Listing 9).

As to the ElevatorSim process. According to the `up` and `down` signals it calculates the elevator coordinate and limits it to the allowed values reflecting the height of the building. When the elevator coordinate is within the range of one of the floors, the process holds the corresponding floor sensor (`onfloor0`, `onfloor1`, `onfloor2`) to true.

```
PROCESS Init
  STATE begin
    START PROCESS Door0Sim;
    START PROCESS Door1Sim;
    START PROCESS Door2Sim;
    START PROCESS ElevatorSim;
    STOP;
  END_STATE
END_PROCESS
```

Listing 8. The poST code of the Init process.

```
PROCESS Door0Sim
  VAR CONSTANT
    DOOR_SPEED : REAL := 0.5;
    DOOR_OPEN_COORD : REAL := -50;
  END_VAR
  VAR
    coord : REAL := 0.0;
  END_VAR
  STATE check_open_close LOOPED
    IF open0 THEN
      coord := coord - DOOR_SPEED;
    ELSE
      coord := coord + DOOR_SPEED;
    END_IF
    IF coord >= 0.0 THEN
      coord := 0.0;
    END_IF
    IF coord <= DOOR_OPEN_COORD THEN
      coord := DOOR_OPEN_COORD;
    END_IF
    IF coord = 0.0 THEN
      door0closed := TRUE;
    ELSE
      door0closed := FALSE;
    END_IF
  END_STATE
END_PROCESS
```

Listing 9. The poST code of the Door0Sim process.

D. CONTROLLER IMPLEMENTATION IN POST

The controller program (Fig. 3) contains 13 processes, combined into two groups. The first group of auxiliary processes consists of processes serving the call buttons and floor LEDs.

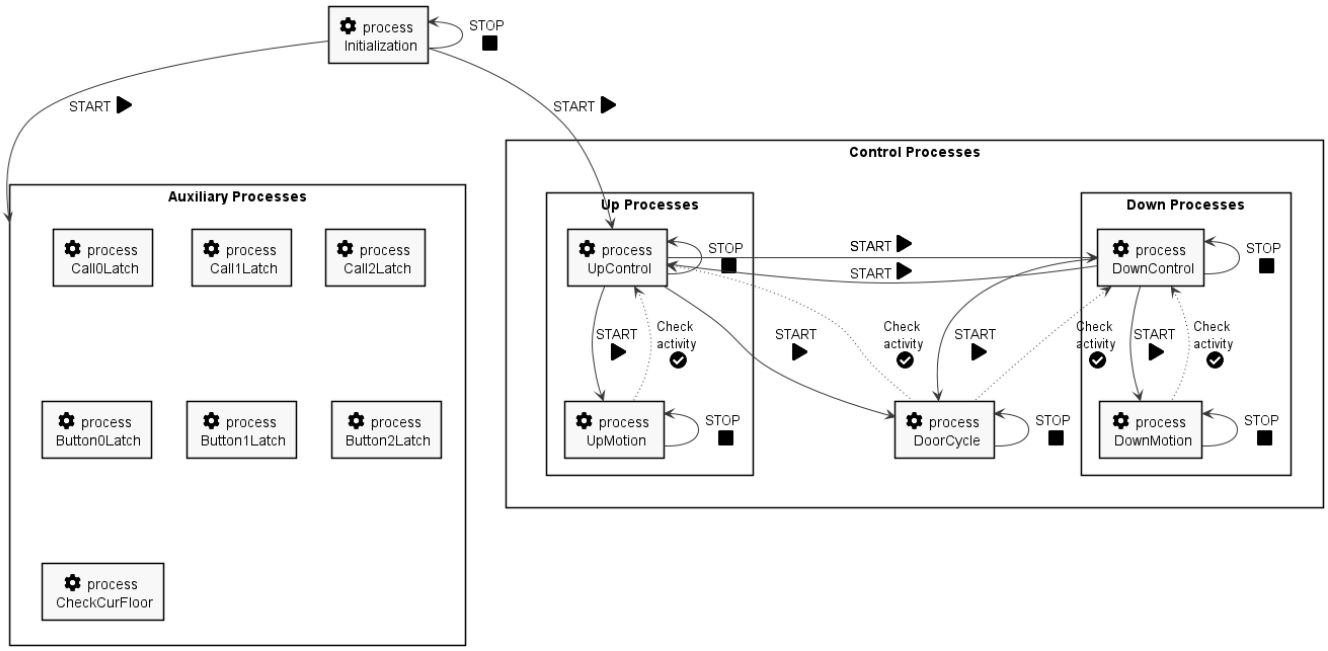


FIGURE 3. Controller process diagram.

```
PROCESS Call0Latch
  VAR
    prev_in : BOOL;
    prev_out : BOOL;
  END_VAR
  STATE init
    prev_in := NOT call0;
    prev_out := NOT open0;
    SET NEXT;
  END_STATE
  STATE check_ON_OFF LOOPED
    IF call0 AND NOT prev_in THEN //rising edge
      call0_LED := TRUE;         // switch on
    END_IF
    IF open0 AND NOT prev_out THEN //rising edge
      call0_LED := FALSE;         // switch off
    END_IF
    prev_in := call0; // for the edges detection
    prev_out := open0;
  END_STATE
END_PROCESS
```

Listing 10. The poST code of the Call0Latch process.

```
  STATE check_floor
    IF onfloor0 THEN
      cur := 0;
      floor0_LED := TRUE;
      floor1_LED := FALSE;
      floor2_LED := FALSE;
    ELSIF onfloor1 THEN
      cur := 1;
      floor0_LED := FALSE;
      floor1_LED := TRUE;
      floor2_LED := FALSE;
    ELSIF onfloor2 THEN
      cur := 2;
      floor0_LED := FALSE;
      floor1_LED := FALSE;
      floor2_LED := TRUE;
    END_IF
  END_STATE
END_PROCESS
```

Listing 11. The poST code of the CheckCurFloor process.

The second group (control processes) consists of the processes of moving the doors and elevator car. The initial process (Initialization process) launches the auxiliary processes and the UpControl process. After this, the process stops itself.

The latch processes (`Call0Latch`, `Call1Latch`, `Call2Latch`, `Button0Latch`, `Button1Latch`, `Button2Latch`) differ only in variables (Listing 10). Each latch process has two local variables and two states. The local variables store previous values of the `call0` and `open0` 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 button press and door opening events via `call0`and `open0` signals. On the button press event it sets the `call0_LED` signal. On the door opening event the process turn off the LED. The `CheckCurFloor` process (Listing 11) tracks the floor sensor signals `onfloor0`, `onfloor1`, `onfloor2`, sets the corresponding LEDs, and stores the current floor number in the `cur` internal variable.

The `UpControl` process (Listing 12) gives priority to calls from upper floors. If the call is on the current floor, the process launches a door opening sequence (`DoorCycle`process) and changes its current state to `door_cycle`. If calls on upper floors are present, it initiates elevator movement in the upward direction by starting the `UpMotion` process and changes its current state to `check_stop`. Otherwise, if calls are from lower floors only, it starts the `DownControl` process and stops itself. In the `door_cycle` state the process waits for the `DoorCycle` process (Listing 13) to stop and then resumes operation in the `check_calls` state.In the `check_stop` state the process awaits arrival to the target floor by monitoring inactive state of the `UpMotion` process. It then starts the `DoorCycle` process and proceeds to the `door_cycle` state.

```
PROCESS UpControl (* up motion priority *)
  STATE check_calls //call from current floor?
    IF (cur = 0 AND (call0_LED OR button0_LED)) OR
    (cur = 1 AND (call1_LED OR button1_LED)) OR
    (cur = 2 AND (call2_LED OR button2_LED)) THEN
      START PROCESS DoorCycle;
      SET STATE door_cycle;
    ELSE // are there other calls?
      CASE (cur) OF
        0: // call from upper floor?
          IF ((call1_LED OR button1_LED) OR
          (call2_LED OR button2_LED))THEN
            START PROCESS UpMotion;
            SET NEXT;
          END_IF
        1: // above?
          IF (call2_LED OR button2_LED) THEN
            START PROCESS UpMotion;
            SET NEXT;
          // below?
          ELSIF (call0_LED OR button0_LED) THEN
            START PROCESS DownControl;
            STOP;
          END_IF
        2: // switch direction
          START PROCESS DownControl;
          STOP;
        END_CASE
    END_IF
  END_STATE
  STATE check_stop
    IF (PROCESS UpMotion IN STATE INACTIVE) THEN
      START PROCESS DoorCycle;
      SET NEXT;
    END_IF
  END_STATE
  STATE door_cycle
    IF (PROCESS DoorCycle IN STATE INACTIVE) THEN
      RESTART; // set the initial state
    END_IF
  END_STATE
END_PROCESS
```

Listing 12. The poST code of the UpControl process.

```
PROCESS DoorCycle
  STATE choose_door_to_open
    CASE (cur) OF
      0:  open0 := TRUE;
      1:  open1 := TRUE;
      2:  open2 := TRUE;
      END_CASE
    SET NEXT;
  END_STATE
  STATE delay3s
    TIMEOUT T#3s THEN
      open0 := FALSE;
      open1 := FALSE;
      open2 := FALSE;
      SET NEXT;
    END_TIMEOUT
  END_STATE
  STATE check_closed
    IF (door0closed AND
      door1closed AND
      door2closed) THEN
      STOP;
    END_IF
  END_STATE
END_PROCESS
```

Listing 13. The poST code of the DoorCycle process.

The `UpMotion` process (Listing 14) starts upward motion, then tracks the active call from the next floor. If an active call is present it sets the target floor and changes its state to `check_target`. In this state it detects arrival to the target floor, then terminates the elevator movement and stops itself. The `DownControl` and `DownMotion` processes implement the same logic for the downward case.

```
PROCESS UpMotion
  STATE start // check the next floor call
    up := TRUE;
    CASE (cur) OF
      0:
        IF (call1_LED OR button1_LED) THEN
          target := 1; // if next floor
          SET NEXT;
        END_IF
      1:
        IF (call2_LED OR button2_LED) THEN
          target := 2; // if next floor
          SET NEXT;
        END_IF
      2:
        target := 2; // a stub
        SET NEXT;
      END_CASE
  END_STATE
  STATE check_target
    IF (cur = target) THEN  // have arrived?
      up := FALSE;
      STOP;
    END_IF
  END_STATE
END_PROCESS
```

Listing 14. The poST code of the UpMotion process.

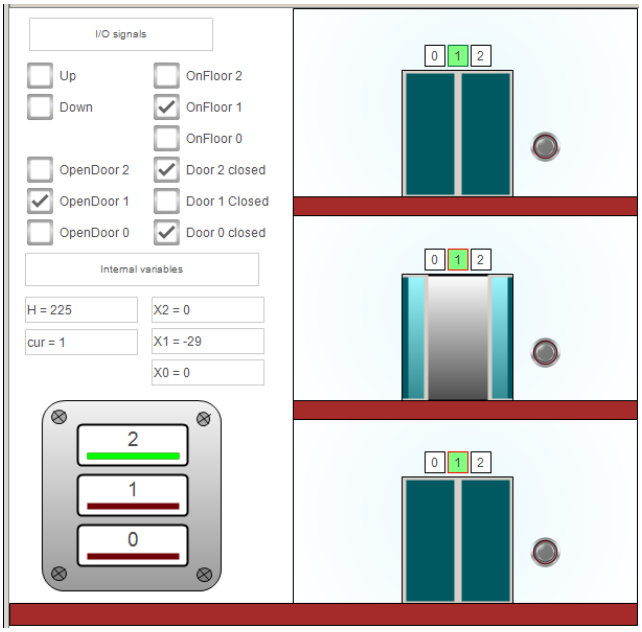


FIGURE 4. CoDeSys visualization of the elevator.

E. DEPLOYMENT AND VISUALISATION IMPLEMENTATION IN CODESYS

The presented control algorithm along with the three-level elevator model was implemented using the poST/Eclipse IDE [66]. The behavior of the control system was dynamically verified using CoDeSys (Fig. 4).

The Controller program implements the control algorithm and is targeted to be uploaded to PLC. The Plant program is a digital twin of the elevator and simulates physical processes within the Plant. It implements the Reconfiguration and Transformation block (Fig. 1) to allow testing of the control algorithm.

The behavior of the control system was dynamically verified using CoDeSys simulation mode [CoDeSys] (Fig. 4).

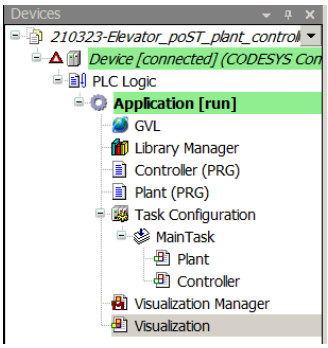


FIGURE 5. CoDeSys elevator project tree.

Here the interaction between the Plant and the Controller is organized via global variables stored in a separate GVL file.

According to this schema, the controller sets the output variables (up/down and open door motors). The Plant simulator reads these variables, then recalculates elevator coordinates in case of motion, and forms input signals such as states of the floor and door sensors. Among the input and output signals, there are controls and indicators (call buttons and LEDs) that implement an interface with the user. When used in actual physical PLC and plant, these global variables would be mapped to PLC input/output ports and represent the PLC input/output signals. The state of the elevator and its user interface are displayed using CoDeSys visualization tools (Fig. 5).

VI. DISCUSSION AND CONCLUSION

In this work a novel programming language poST – a process-oriented extension of IEC 61131-3 Structured Text is presented. poST assumes specification of control algorithms as a set of communicating FSM-based processes. The hyper-process model underlying poST is compliant with primary features of control software – cyclic execution, event-driven and polymorphic behavior, temporal behavior, and concurrency. The language syntax is similar to the IEC 61131-3 Structured Text and provides time-interval operations as well as means for inter-process communication that encourage a hierarchical organization of control software. A translator from poST to ST has been implemented and semantics of key poST constructs has been demonstrated. The case study demonstrates promising qualities of the resulting code such as readability, maintainability, and overall robustness. This shows that the language can be successfully used in complex industrial applications.

Within the process-oriented approach, it provides the following advantages.

1) The syntax of the language enables explicit specification of state machines, which ST programmers use for complex control algorithms. Compared to state machine description using bare ST, this reduces code size and has a positive effect on readability. Furthermore, this enables automatic semantic checks in high-level control domain terms such as interaction between processes.
2) poST encourages control algorithm specification to be correspondent to the technological description of processes inside the controlled plant. This allows static analysis tools to provide a high-level graphical representation of the source code with UML and process diagrams.
3) poST is a textual language, which is an attractive feature for many developers, as compared to the graphical notations of PLC programming, such as SFC.

Therefore, the poST language enables an application-centric approach to control software development and improves the maintainability (readability, error traceability, changeability, etc.) of the control software.

Apart from the above-mentioned advantages, the language can employ formal verification and code analysis methods earlier developed for Reflex, due to the similarity of the two languages.

Experimental study of the poST language with the three-level elevator example has shown that developing programs in poST can yield groups of highly similar processes. One kind of such processes (e. g., the latch processes) differ only in their input and output signals. The other kind, such as UpControl and DownControl, differ in their used child processes. Description of these processes can be reduced using some sort of process templating.

Further direction of work would be to develop templating means and also adapt the IEC 61131-3 configuration mechanism for poST.

ACKNOWLEDGMENT

The authors thank the JetBrains Foundation for the charitable support of our research activity.

REFERENCES

[1] *Programmable Controllers—Part 3: Programming Languages*, document IEC 61131-3, International Electrotechnical Commission, 2013.
[2] F. Basile, P. Chiacchio, and D. Gerbasio, "On the implementation of industrial automation systems based on PLC," *IEEE Trans. Autom. Sci. Eng.*, vol. 10, no. 4, pp. 990–1003, Oct. 2013, doi: 10.1109/TASE.2012.2226578.
[3] K. C. Crater, *When Technology Standards Become Counterproductive*. Hopkinton, MA, USA: Control Technology Corporation, 1992. [Online]. Available: https://support.controltechnologycorp.com/index.php?option=com_content&view=article&id=188&Itemid=null
[4] F. Wagner. *StateWORKS: Going beyond the limitations of IEC 61131-3*. Shreveport, LO, USA: StateWORKS, 2005. [Online]. Available: https://www.stateworks.com/active/download/TN11-Going-Beyond-Limitations-Of-IEC-61131.pdf
[5] K. Thramboulidis and G. Frey, "Towards a model-driven IEC 61131-based development process in industrial automation," *J. Softw. Eng. Appl.*, vol. 4, no. 4, pp. 217–226, 2011.
[6] U. Katzke and B. Vogel-Heuser, "Combining UML with IEC 61131–3 languages to preserve the usability of graphical notations in the software development of complex automation systems," *IFAC Proc.*, vol. 40, no. 16, pp. 90–94, 2007.
[7] B. Werner, "Object-oriented extensions for IEC 61131," *IEEE Ind. Electron. Mag.*, vol. 3, no. 4, pp. 36–39, Oct. 2009.
[8] W. Pessemier, G. Deconinck, G. Raskin, P. Saey, and H. Van Winckel, "Developing a PLC-friendly state machine model: Lessons learned," *Softw. Cyberinfrastruct. Astron.*, vol. 9152, Oct. 2014, Art. no. 915208.
[9] *International Standard IEC61499-1: Function Blocks—Part 1 Architecture*, International Electrotechnical Commission, Geneva, Switzerland, 2012.

[10] H. Berger, *Automating With SIMATIC: Integrated Automation With SIMATIC S7-300/400: Controllers, Software, Programming, Data Communication, Operator Control and Process Monitoring*. Publicis Corporate, 2006.

[11] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme, *Modeling Software with Finite State Machines*. Boston, MA, USA: Auerbach, 2006.

[12] M. Samek, *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. Oxford, U.K.: Newnes, 2009.

[13] A. Shalyto and N. Tukkel, "SWITCH technology: An automated approach to developing software for reactive systems," *Program. Comput. Softw.*, vol. 27, no. 5, pp. 260–276, 2001.

[14] B. Yartsev, G. Korneev, A. Shalyto, and V. Kotov, "Automata-based programming of the reactive multi-agent control systems," in *Proc. Int. Conf. Integr. Knowl. Intensive Multi-Agent Syst.*, Apr. 2005, pp. 449–453.

[15] V. Zyubin, "SPARM language as a means for programming microcontrollers," *Optoelectron., Instrum., Data Process.*, vol. 2, pp. 36–44, May 1996.

[16] D. Witsch and B. Vogel-Heuser, "PLC-statecharts: An approach to integrate UML-statecharts in open-loop control engineering-aspects on behavioral semantics and model-checking," *IFAC Proc.*, vol. 44, no. 1, pp. 7866–7872, 2011.

[17] *Data sheet CODESYS Professional Developer Edition—Product Data Sheet V0.0.0.1*, CODESYS GmbH, Kempten, Germany, 2019, pp. 1–4.

[18] H. Dierks, "PLC-automata: A new class of implementable real-time automata," in *Transformation-Based Reactive System Development*, vol. 1231, M. Bertran and T. Rus, Eds. Berlin, Germany: Springer-Verlag, 1997, pp. 111–125.

[19] N. Hagge and B. Wagner, "Implementation alternatives for the OMAC state machines using IEC 61499," in *Proc. IEEE Int. Conf. Emerg. Technol. Factory Autom.*, Sep. 2008, pp. 215–220, doi: 10.1109/etfa.2008.4638395.

[20] H. Jack, "Automating manufacturing systems with PLCs," Lulu.com, 2010.

[21] D. Darvas, E. B. Vinuela, and I. Majzik, "PLC code generation based on a formal specification language," in *Proc. IEEE 14th Int. Conf. Ind. Informat. (INDIN)*, Jul. 2016, pp. 389–396.

[22] T. K. Tran, H. Yahoui, and N. Siauve, "An interactive approach to teach automation in the training of the industry 4.0," in *Proc. 13th Int. Conf. Softw., Knowl., Inf. Manage. Appl. (SKIMA)*, Aug. 2019, pp. 1–4, doi: 10.1109/skima47702.2019.8982491.

[23] S. Henneken, *The 'State' Pattern*, document IEC 61131-3, 2018. [Online]. Available: https://stefanhenneken.net/2018/11/17/iec-61131-3-the-state-pattern/

[24] J. Muhammad, *PLC Programming an Elevator With Structured Text in Codesys*. Philadelphia, PA, USA: Gallop Automation Blog, 2020. [Online]. Available: https://blog-gallopautomation.com/plc-programming-an-elevator-with-structured-text-in-codesys_part1modular-approach/

[25] D. Gritzner and J. Greenyer, "Synthesizing executable PLC code for robots from scenario-based GR (1) specifications," in *Proc. Fed. Int. Conf. Softw. Technol., Appl. Found.*, Jul. 2017, pp. 247–262.

[26] D. Harel, "StateChart: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, pp. 231–274, Oct. 1987.

[27] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, "Toward a lingua franca for deterministic concurrent systems," *ACM Trans. Embedded Comput. Syst.*, vol. 20, no. 4, pp. 1–27, 2021.

[28] K. Crater, "A white paper state language for machine control," in *Proc. Control Technol.*, Hopkinton, MA, USA, 1999, pp. 1–11.

[29] R. Alur and D. Dill, "Automata for modeling real-time systems," in *Proc. Int. Colloq. Algorithms, Lang., Program.*, vol. 443, 1990, pp. 322–335.

[30] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, pp. 194–1955, Dec. 1994.

[31] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL—A tool suite for automatic verification of real-time systems," in *Proc. Int. Hybrid Syst. Workshop*, 1995, pp. 232–243.

[32] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A model-checking tool for real-time systems," in *Proc. Int. Symp. Formal Techn. Real-Time Fault-Tolerant Syst.*, Sep. 1998, pp. 298–302.

[33] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[34] R. H. Hardin, Z. Har'El, and R. P. Kurshan, "Cospan," in *Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 1996, pp. 423–427.

[35] K. McMillan, "The SMV system," in *Proc. Symbolic Modeling Checking*, 1993, pp. 61–85.

[36] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. Mckenzie, "SMV—Symbolic model checking," in *Proc. Syst. Softw. Verification*, 2001, pp. 131–138, doi: 10.1007/978-3-662-04558-9_12.

[37] J. Perna and G. Chris, "Model checking RAISE specifications," UNU-IIST, Macau, Asia, Tech. Rep. 331, 2005.

[38] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, "Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems," in *Proc. Int. Symp. Syst.-Chip*, Cancun, Mexico,2003, pp. 166–177.

[39] K. Sacha, "Automatic code generation for PLC controllers," in *Int. Conf. Comput. Saf., Rel., Secur.*, pp. 303–316, 2005.

[40] V. E. Zyubin, "Hyper-automaton: A model of control algorithms," in *Proc. Siberian Conf. Control Commun.*, Apr. 2007, pp. 51–57, doi: 10.1109/SIBCON.2007.371297.

[41] A. S. Rozov and V. E. Zyubin, "Adaptation of the process-oriented approach to the development of embedded microcontroller systems," *Optoelectron., Instrum. Data Process.*, vol. 55, no. 2, pp. 198–204, Mar. 2019.

[42] V. Vyatkin, H. M. Hanisch, C. Pang, and C. H. Yang, "Closed-loop modeling in future automation system engineering and validation," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 39, no. 1, pp. 17–28, Jan. 2009.

[43] A. Rajhans, S. Avadhanula, A. Chutinan, P. J. Mosterman, and F. Zhang, "Graphical modeling of hybrid dynamics with simulink and stateflow," in *Proc. 21st Int. Conf. Hybrid Syst., Comput. Control*, Apr. 2018, pp. 247–252.

[44] C. Secchi, M. Bonfe, C. Fantuzzi, R. Borsari, and D. Borghi, "Object-oriented modeling of complex mechatronic components for the manufacturing industry," *IEEE/ASME Trans. Mechatronics*, vol. 12, no. 6, pp. 696–702, Dec. 2007.

[45] K. C. Thramboulidis, "Using uml in control and automation: A model driven approach," in *Proc. 2nd Int. Conf. Ind. Informat.*, 2004, pp. 587–593.

[46] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State of the art review," *IEEE Trans. Ind. Informat.*, vol. 7, no. 11, pp. 768–781, Nov. 2011.

[47] P. Jhunjhunwala, U. Atmojo, and V. Vyatkin, "Towards implementation of interoperable smart sensor services," in *Proc. 25th IEEE Int. Conf. Emerg. Technol. Factory Autom.*, vol. 1, Sep. 2020, pp. 1409–1412.

[48] D. Drozdov, U. D. Atmojo, C. Pang, S. Patil, M. I. Ali, A. Tenhunen, T. Oksanen, K. Cheremetiev, and V. Vyatkin, "Utilizing software design patterns in product-driven manufacturing system: A case study," in *Proc. 9th Int. Workshop Service Oriented, Holonic Multi-Agent Manuf. Syst. Ind. Future, Stud. Comput. Intell.*, vol. 853, 2019, pp. 301–312.

[49] S. Patil, D. Drozdov, and V. Vyatkin, "Adapting software design patterns to develop reusable IEC 61499 function block applications," in *Proc. IEEE 16th Int. Conf. Ind. Informat. (INDIN)*, Jul. 2018, pp. 725–732.

[50] G. Frey, "Automatic implementation of Petri net based control algorithms on PLC," in *Proc. Amer. Control Conf.*, 2000, pp. 2819–2823.

[51] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An operating system for sensor networks," in *Proc. Ambient Intell.*, 2005, pp. 115–148.

[52] D. Gay, Ph. Levis, D. Culler, and E. Brewer. (2003). *NesC 1.1 Language Reference Manual*. [Online]. Available: http://nescc.sourceforge.net/papers/nesc-ref.pdf

[53] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proc. ACM SIGPLAN Conf. Program. Lang. design Implement.*, 2003, pp. 1–11.

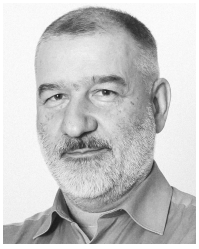
[54] D. Bulavskij, V. Zyubin, N. Karlson, V. Krivoruchko, and V. Mironov, "An automated control system for a silicon single-crystal growth furnace," in *Proc. Optoelectron., Instrum., Data Process.*, vol. 2, 1996, pp. 25–30.

[55] I. Anureev, N. Garanina, T. Liakh, A. Rozov, H. Schulte, and V. Zyubin, "Towards safe cyber-physical systems: The reflex language and its transformational semantics," in *Proc. Int. Siberian Conf. Control Commun. (SIBCON)*, Apr. 2019, pp. 1–6.

[56] *QuickstepTM Language and Programming Guide*, document MAN-1010A, Control Technology Corporation, 2000. [Online]. Available: http://www.ctc-control.com/customer/techinfo/docs/QuickstepLangProg.pdf

[57] A. Rozov, I. Anureev, N. Garanina, T. Liakh, and V. Zyubin, "Towards safe embedded systems: Industrial C translational semantics for AVR Microcontrollers," in *Proc. Int. Multi-Conf. Eng., Comput. Inf. Sci. (SIBIRCON)*, Oct. 2019, pp. 857–861, doi: 10.1109/SIBIRCON48586.2019.8958258.

[58] H. Hanisch, "Closed-loop modeling and related problems of embedded control systems in engineering," in *Proc. Abstract State Mach., Adv. Theory Pract.*, Lutherstad, Wittenberg, 2004, pp. 24–28.
[59] I. Petrov and R. Wagner, "Debugging applied PLC software in CoDeSys," in *Proc. Ind. Autom. Control Syst. Controllers*, 2006, pp. 34–36.
[60] V. Zyubin, "Multicore processors and programming," *Open Syst. J.*, vol. 4, nos. 7–8, pp. 12–19, 2005.
[61] V. Zyubin, "Text and graphics: What language does programmer need?" *Open Syst. J.*, vol. 1, pp. 54–58, Dec. 2004.
[62] V. Glushkov, *The Synthesis of Digital Automata*. Moscow, Russia: PhisMathGis, 1962.
[63] F. Wagner and P. Wolstenholme, "Misunderstandings about state machines," *Comput. Control Eng.*, vol. 15, no. 4, pp. 40–45, Aug. 2004.
[64] L. Kof and B. Schätz, "Combining aspects of reactive systems," in *Proc. Andrei Ershov Fifth Int. Conf. Perspect. Syst. Inform.*, Novosibirsk, Russia, 2003, pp. 239–243.
[65] Festo Didactic GmbH. (2003). *EasyVeep Handbuch Manual*. [Online]. Available: https://www.festo-didactic.com/ov3/media/customers/1100/00730997001075223738.pdf
[66] V. Bashev, A. Rozov and V. Zyubin, "PoST2ST: A web service for translating post programs to the IEC 61131-3 structured text," in *Proc. IEEE 22nd Int. Conf. Young Prof. Electron Devices Mater. (EDM)*, Oct. 2021, pp. 520–523, doi: 10.1109/EDM52169.2021.9507695.



VLADIMIR E. ZYUBIN received the Ph.D. degree in applied computer science from the Institute of Automation and Electrometry SB RAS, Russia, in 1998, and the Dr.Sc. degree from the Siberian State University of Telecommunications and Information Sciences, Russia, in 1998.

He is currently on joint appointment as the Chair of computer technology at Novosibirsk State University, Russia; and the Head of the Cyber-Physical Laboratory, Institute of Automation and Electrometry. His research interests include process-oriented programming and languages for PLC control, software psychology, requirements engineering for control systems, and verification of reactive software.

ANDREI S. ROZOV received the master's degree from the Department of Information Technologies, Novosibirsk State University (NSU), in 2012, and the Ph.D. degree in applied computer science from the Institute of Automation and Electrometry SB RAS, Russia, in 2021.

He is currently teaching courses on embedded systems and microcontroller programming for NSU students and researching the field as part of Prof. Zyubin's team at the Laboratory of Cyber-Physical Systems, Institute of Automation and Electrometry SB RAS. He has participated in and led multiple projects in embedded systems and programming languages development. His research interests include embedded software design methods, microcontroller programming, domain-specific programming languages, program semantics, source code ergonomics, code analysis, and reverse engineering.

IGOR S. ANUREEV was born in Novosibirsk, in 1971. He received the M.S. degree in computer science from Novosibirsk State University, Novosibirsk, in 1994, and the Ph.D. degree in computer science from the A. P. Ershov Institute of Informatics Systems SB RAS, Novosibirsk, in 1998.

He has been a Senior Researcher at the Laboratory of Theoretical Programming, A. P. Ershov Institute of Informatics Systems, since 2004; and a Senior Researcher at the Cyber-Physical Systems Laboratory, Institute of Automation and Electrometry SB RAS, since 2019. He is the author of more than 100 articles. His research interests include specification and verification of program systems and program models, design and prototyping of program systems and program models, semantics of program systems and computer languages, domain-specific languages, automated theorem proving, and ontologies.

NATALIA O. GARANINA received the M.S. degree in computer science from Novosibirsk State University, Novosibirsk, in 2001, and the Ph.D. degree in computer science from the A. P. Ershov Institute of Informatics Systems SB RAS, Novosibirsk, in 2004.

She has been a Senior Researcher at the Laboratory of Theoretical Programming, A. P. Ershov Institute of Informatics Systems, since 2014; a Senior Researcher at the Cyber-Physical Systems Laboratory, Institute of Automation and Electrometry SB RAS, since 2019; and a Lecturer at Novosibirsk State University, since 2007. She is the author of more than 100 articles. Her research interests include formal verification, distributed systems, automatic control systems, artificial intelligence, non-classical logics, and domain theory.

VALERIY VYATKIN (Fellow, IEEE) received the Ph.D. and Dr.Sc. degrees in applied computer science from the Taganrog Radio Engineering Institute, Russia, in 1992 and 1999, respectively, the Dr.Eng. degree from the Nagoya Institute of Technology, Japan, in 1999, and the Habilitation degree from Germany, in 2002. He was a Visiting Scholar with Cambridge University, U.K. He had permanent appointments with The University of Auckland, New Zealand; and Martin Luther University, Germany; as well as in Japan and Russia. He is currently on joint appointment as the Chair of dependable computations and communications at the Luleå University of Technology, Sweden; and a Professor of information technology in automation with Aalto University, Finland. He is also the Co-Director of the International Research Laboratory "Computer Technologies," ITMO University, Saint Petersburg, Russia. His research interests include dependable distributed automation and industrial informatics, software engineering for industrial automation systems, artificial intelligence, distributed architectures, and multi-agent systems in various industries, including smart grid, material handling, building management systems, data centers, and reconfigurable manufacturing. He has received the Andrew P. Sage Award for the Best IEEE TRANSACTIONS Paper, in 2012. He is the Chair of IEEE IES Technical Committee on Industrial Informatics.