УДК 004.432:004.054

Reflex Language: a Practical Notation for Cyber-Physical Systems

Liakh T.V., Rozov A.S., Zyubin V.E.

(Institute of Automation and Electrometry SB RAS,

Novosibirsk State University)

This paper introduces a conceptual framework for complex control algorithms in form of hyperprocess model. To demonstrate the practical value of the model we describe grammar and translational semantics of a process-oriented language, known as Reflex or "C with processes". Expressive properties of the presented notation are shown on an example control algorithm design for a hand dryer device. Finally, we give a short report about practical application of the language and results, which have been obtained during its usage.

Keywords: Reflex language, control software, IEC 61131-3, process-oriented programming, POP, programmable logic controller, PLC, cyber-physical system, hybrid system, finite state machine, FSM, finite state automaton, FSA, controller, plant, syntax, semantics, industrial automation, reactive system

1. Introduction

Industrial control systems are nowadays most commonly implemented using programmable logic controllers (PLCs). A typical PLC consists of a central control board hosting the processor, in bundle with multiple extension modules that provide digital I/O, analog-to-digital and digital-to analog conversions, etc. All of control logic, including timing, computation, continuous control and peripheral communication, is specified in the PLC software. Thus in control systems, software development expenses surpass those of hardware development and constitute the larger part of the total project cost.

The specificity of control algorithms calls for specialized design patterns, languages and models to be utilized in development of PLC software. Numerous theoretical studies have been published, with design patterns, models and methods (e. g. TCSP, CCS, I/O-Automata, TLA, FSM and their timed extensions [1–12], to name a few) that address the characteristic issues arising in control software development. Despite this diversity of approaches available, the industry still favours IEC 61131-3 [13] standard languages as their primary and ultimate solution

in the field. It incorporates languages for relay logic (LD), functional block diagrams (FBDs), sequential function charts (SFCs) and assembler-like specifications (IL). However, as the complexity of control software increases and quality is of higher priority, the 35 years old technology underlying the IEC 61131-3 approach is not able to address the present-day requirements [14]. The IEC standard is ambiguous, universally critisized for its lack of expressiveness and is even cited as an example of standardization misuse [15]. The shortcomings of the IEC 61131-3 languages have become particularly evident after new types of cyber-physical systems started to emerge and rapidly spread, such as embedded systems, smart-devices, IoT-devices, etc. This motivates researchers to enrich the IEC 61131-3 development model with OO concepts [16], or rather develop alternative approaches, e. g. [17, 18].

The idea of designing special mathematical models for specification of control software is not new. A large amount of related work has been published over the years. Unfortunately, the effort was distributed among different fields: reactive systems, programming, logic, parallel system, homeostatic systems, so called real time systems, robotics, etc., see e. g. [1, 4, 5, 9, 19] Though meany of the presented ideas look promising, these theoretical results find little to no use in the industry. Attaining a practical solution for control system specification requires a holistic approach.

We strongly believe that a useful notation for control software can be designed over a thorough analysis of the target task. The models and concepts underlying such a notation would have to reflect the key features of control systems. Further, the notation itself should be comprehensible and easy-to-use for the programmers already engaged in the industry. It therefore needs to conform with the current trends in programming languages. In this work we introduce a programming language based on this idea.

The paper is organized as follows: in Section 2 we analyze the domain of control systems and extract the key features of control software. Section 3 introduces the hyperprocess mathematical model that reflects the aforementioned features. Section 4 presents the syntax of Reflex programming language that is based on the hyperprocess. Section 5 shows semantics of the language. Section 7 concludes the paper and lines out directions for our future work.

This work has been supported by the Federal Agency for Scientific Organizations (project AAAA-A17-117060610006-6) and the Russian Foundation for Basic Research (grant 17-07-01600).

2. Specific Features of Control Software

When considering a modern control system, we generally picture a digital controller connected to a controlled plant. The plant represents external environment of 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 on them with accordance to its algorithm. 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 current state of controlled plant from sensors – data processing and determination of required reaction – and data output, which changes 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 dynamical systems, control software has to function time-dependently, i. e. accordingly 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 leads to a combinatorial explosion of complexity [20]. 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 artificial nature of the external environment, the designer plan that is implemented in architecture of the facilities [21]. 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 control algorithm. The algorithm structure can be arbitrary, irregular and ever closed on itself.

To summarize our analysis, we list the key features of control systems that we deem most significant for control software development:

- openness – i. e. communication with an external environment,
- cyclic and indefinite execution,
- event-driven polymorphic behavior,
- operations with time intervals,
- concurrency,
- hierarchical structure.

3. Hyperprocess Model

From practice it is apparent that the Finite State Automaton (FSA) concept is particularly promising for use in logical control. FSAs implicitly assume 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 FSA model is tailored for hardware implementation. This is due to historical circumstances of when the model was created [22]. 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 a potentially very powerful concept [23]. Efficiency in practice requires that more conventional programming concepts, like variables and statements be supported. With this in mind, we have constructed an FSA-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 Reflex language. A more detailed description of the hyperprocess model can be found in [24].

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 programming sense). Note that the word "ordered" refers to textual description of a program. It should also be noted that a kind of perfect synchrony hypothesis [25] 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 or equal than 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 hyper-process 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 FSA 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 FSA model was preserved within the process model. Describing software with multiple automata provides logically parallel execution with granularity of the functions. The hyperprocess execution can be described in a 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 process 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 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 FSA model, and extended it to support concurrency, hierarchy and time tracking.

4. Introduction to Reflex Syntax

The hyperprocess model presented above, underlies the Reflex programming language, which is a dialect of the C programming language. The C language has been chosen for better learnability as many mainstream programming languages nowadays have C-like syntax. Additional constructs have been introduced for controlling processes and tracking time intervals. Two kinds of passive functions are used: the STOP-function is for a normal finishing of the process, and the ERROR-function is to indicate an abnormal finish. Reflex also provides constructs for linking internal software variables to physical I/O signals. The direct and inverse transformations between registers of I/O modules and internal variables are automated.

Reflex syntax is demonstrated here using a simple example of a program controlling a hand dryer like those often found in public restrooms (Listing 1). A formal Reflex syntax definition in EBNF can be found in Appendix A.

Here, the program uses input from an IR sensor, indicating presence of hands under the dryer and controls the fan and heater with a joint output signal. The basic requirement is that the dryer is on while hands are present and turns off automatically otherwise. As the person using the dryer rubs and turns their hands, the signal from the binary IR sensor will pulse between "on" and "off". To avoid erratic toggling of the dryer heater and fan, the program should not react to this switching and the actuators should only be turned off once the sensor signal is a steady "off". The algorithm can meet such requirement only by measuring the duration of the off state of the sensor. In this case, a duration longer than a certain given value (for example, 1 s) would be regarded as the "hands removed" event.

```
PROGR HandDryerController {
  TACT 100;
  CONST ON 1;
  CONST OFF 0;
  /* direction, name, address, offset, size of the port */
  INPUT  SENSOR_INPUT_PORT  0 0 8; /*IR sensor input port*/
  OUTPUT ACTUATOR_OUTPUT_PORT 1 0 8; /*output to heater and fan*/
  PROC Init {
    BOOL S_HANDS_UNDER_DRYER = {SENSOR_INPUT_PORT[1]} FOR ALL;
    BOOL C_TURN_ON_DRYER = {ACTUATOR_OUTPUT_PORT[1]} FOR ALL;
    STATE Waiting {
      IF (S_HANDS_UNDER_DRYER == ON) {
        C_TURN_ON_DRYER = ON;
        SET NEXT;
      }
      ELSE C_TURN_ON_DRYER = OFF;
    }
    STATE Drying {
      IF (S_HANDS_UNDER_DRYER == ON) RESET TIMEOUT;
      TIMEOUT 10 { SET STATE Waiting; }
    }
  } /* \PROC */
}  /* \PROGRAM */
```

Listing 1: Hand dryer example in Reflex

General program structure. A Reflex program is an aggregate of concurrently running communicating processes defined in textual form with the `PROC` keyword:

```
PROC <process name> { <process body> }
```

The process that is defined first in program text is the process p_1 that is initially in active state upon start of the program.

The `TACT` directive at the top sets the hyperprocess activation period T_H specified in milliseconds.

Main part of the process body is a list of state function definitions:

```
STATE <state name> { <state body> }
```

The first defined state in process body corresponds to it starting state f_i^1 into which the process is transitioned by a `START PROC` instruction (or initially for p_1). Passive states `STOP` and `ERROR` are not explicitly defined in the program.

Statements. A state body is defined with a sequential block of code, that includes statements like expression calculations (this include assignments), C-like selection statements `IF/ELSE` and `SWITCH` that define events and their corresponding reactions, process control statements and one optional timeout statement.

The syntax for the selection statements is very similar to that in C:

```
IF (<expression>) <statement> ELSE <statement>
```

Each of the `<statement>` blocks here can either be a single, semicolon terminated statement, or a compound statement enclosed in braces.The `ELSE` part is optional. Nested `IF`s are supported, i.e. each of the `<statements>` blocks above can in turn contain selection statements, much like in C language. The C-like `SWITCH` statement has also been included in Reflex:

```
SWITCH(<expression>) {
    CASE <value>:
        <statement>
    ...
    DEFAULT:
        <statement>
}
```

Reflex-specific process control constructs include state transitions, control statements and activity predicates that can be used in expressions. State transitions set the process state for the next activation cycle:

```
SET STATE <state name>;
```

A reserved keyword `NEXT` can be used here in lieu of explicit state name to denote a transition to the state that is defined next to the current along the text.

The `START/STOP/ERROR` statements allow processes to start/stop other processes and to stop themselves - either normally or in error state. These statements are responsible for divergence and convergence of control flow:

```
START PROC <process name>;
STOP PROC <process name>;
STOP;
ERROR;
```

To provide means for tracking time, timeout statements have been introduced in Reflex:

```
TIMEOUT <value in hyperprocess clocks> <statement>
```

This statement can only be used once in a state function and should then be the last statement in the state body. It allows to specify a reaction to the event of the process spending more than the specified amount of time in its current state. It is worth noting that though the timeout value here is specified in hyperprocess clocks and so the accuracy depends on the hyperprocess activation period T_H defined with the `TACT` directive at the top of the program text.

One use of timeouts is to implement a non-blocking delay, but a more typical application is for when a process waits for some external event, and that event not arriving withing a reasonable time interval would mean a malfunction of external components. Specifying a timeout serves as

a safety mechanism to prevent the process from locking in the waiting state forever, not unlike hardware watchdogs found in microcontrollers. However, it is often the case that some other event can signify that the system is still functioning normally, and the process can stay in its current state for more time. In this situation the `RESET TIMEOUT;` statement can be used.

Expressions. Expressions in Reflex can be used in conditions for `IF` or `SWITCH` selection statements or in expression statements:

```
<expression>;
```

A typical example of this kind of expression statement is assignment of value to a variable. Reflex supports most of expressions found in C, including assignments, comparisons, arithmetic, bitwise and logical operations.

One type of expressions that is new in Reflex is the process activity check predicates. Processes are able to check whether other processes are in their active or passive states using the selection statement in conjunction with `ACTIVE/PASSIVE` predicates, e.g.:

```
IF (PROC <process name> IN STATE ACTIVE) { ... }
```

In the above example, the `IF` condition will be satisfied if the process at question is currently in any state other than `STOP` and `ERROR`.

Ports and variables. The process body can contain variable definitions with port bindings and scope directives:

```
<type> <variable name> = <port binding> <scope directive>;
```

Supported types are `BOOL` for Boolean values as well as `INT`, `SHORT`, `LONG`, `FLOAT` and `DOUBLE` that behave the same way as in C. The `FOR ALL` scope directive is to indicate that this variable can be used by any processes in the program. Port binding makes the variable being read into from an input port or written into the port if that port is defined as output. Ports used in the program are defined before the process definitions in the following format:

```
<direction> <port name> <base address> <offset> <size in bits>;
```

Reflex also supports constant definitions:

```
CONST <constant name> <constant value>;
```

Supported types for constant values here are the same as for variables. Unnamed constants can also be used in expressions, though it is generally deemed a bad practice in programming.

5. Translational Semantics

Semantics of a programming language can be preserved when that language is translated into another (target) programming language [26]. This is especially valuable if the target language is widely used and its semantics are well-known and affixed in standards. A translator has been implemented, that parses programs written in Reflex and produces C code, that can consequently be compiled into executable code for the target platform. C has been chosen as the target language for two reasons. Firstly, C served as the basis for Reflex, and many lower-level Reflex syntax elements have the same meaning as their identical C counterparts. Secondly, the C language is widely spread, used across many application domains and can be easily compiled for almost any known computational platform, thus providing cross-platform compatibility. The semantics of Reflex is therefore defined by this translation and here we use it to demonstrate the meaning of Reflex programs, by providing equivalent C code for key Reflex constructs.

Structure and Execution of an Equivalent C Program

The states of all processes are stored in an array of the following structure:

```
struct StateWord{
  unsigned long T; /*Ticks spent without a state change*/
  unsigned char cur_state; /*Index of current state*/
}ProcStates[PROC_NUM]; /*Array of process states*/
#define STATE_OF_ERROR      254
#define STATE_OF_STOP       255
```

Listing 2: State word structure

Here T corresponds to time T_i in the process model and `cur_state` is the index of current state function: $f_i^{cur} = f_i^j \Rightarrow j - 1 =$ `cur_state`. The first state function f_i^1 has an index of 0 and indices of special states stop and error are defined as the last two indices in the range. Note that `cur_state` is 8-bit here, as typical process in practice has 5 states only and few processes need more than 10 states.

Upon start, the program performs platform-dependent initialization, including that of the time service, which tracks the activation period T_H. Process states are also initialized: all times are set to zero, and all state indices are set to STOP, except for the first process, which is set to be in its first state. After that the program runs in an infinite loop, activating the hyperprocess with period T_H, specified by the `TACT` directive (Listing3). In the presented implementation, the time service is external to Reflex program and its interface is comprised of the `TH_Elapsed` flag.

Alternative implementations could have the infinite loop be empty and hyperprocess activation placed inside a timer interrupt service routine.

```
void main (void){
  SysInit(); /*Platform specific code*/
  InitProcesses(); /*Set p1 to start, the rest to STOP*/
  for(;;){ /*Indefinite cycic execution*/
    if(TH_Elapsed){/*Activation period TH*/
      TH_Elapsed = 0;
      HyperProcess(); /*Activate hyperprocess*/
    }
  }
}
```

Listing 3: Program execution

When activated, the hyperprocess reads and stores all used input port values, consequently executes all processes, writes all output port values that have been changed and, finally, increments time counters of all processes. Together with the previous listing, this makes up a simple round-robin cooperative concurrency algorithm. The reasons behind Input and Output functions are discussed later in the text. `IncrementProcTicks` here increments the `T` values in the `ProcStates` array for all processes. These values are further used in timeout statements.

```
void HyperProcess (void){
  Input(); /*Read and store input port values*/
  Process0(); /*Execute first process*/
  Process1(); /*Execute second process*/
  ...
  ProcessN(); /*Execute last process*/
  Output(); /*Output port values that have changed*/
  IncrementProcTicks(); /*Increase all T values by 1*/
}
```

Listing 4: Hyperprocess execution

Below is the equivalent C code for process `Init` from the hand dryer example demonstrated in 4. It uses a classic switch-style implementation of a finite state automaton. Another common FSA implementation in C involves tabular representation of the state machine with function pointers used for state functions. However, we find that the switch-based version is simpler, provides a more readable C-code and causes less overhead, especially with a significantly limited stack size. Resulting C code could be made even more readable by using enumerators for process and state indices.

```
void Process0 (void) { /*Process Init*/
  switch (ProcStates[0].cur_state) {
    case 0: /*State Waiting*/
```

```
      if (POV0 == C_0) { /*IF(S_HANDS_UNDER_DRYER == 0)*/
        POV1[1] = C_0; /*C_TURN_ON_DRYER = ON;*/
        Set_State(0, 1); /*SET  NEXT;*/
      }
      else POV1[1] = C_1; /*C_TURN_ON_DRYER = OFF;*/
      break;
    case 1: /*State Drying*/
      if (POV0 == C_0) /*IF(S_HANDS_UNDER_DRYER == ON)*/
        Set_State(0, 1); /*RESET TIMEOUT;*/
      if (Timeout(0, 12)) /*TIMEOUT 12*/
        Set_State(0, 0); /*SET STATE Waiting*/
      break;
    default:
      break;
  }
}
```

Listing 5: Example of process execution

Semantics of Special Control Statements and Expressions

Many statements and expressions defined in Reflex, are borrowed from C along with their semantics. Hence, out focus is on those constructs that are unique to Reflex and not present in C. Here we define the semantics of specialized constructs that are necessary for process functioning and inter-process communication, namely the state transition, the start/stop/restart operators, the timeout statement, and the active/passive expressions.

Given this code is found inside a state function of process p_i, state S has index j and process P has index k, and statement <RS> translates to statement <CS> in the resulting code, the following translation rules apply for specialized Reflex statements and logical expressions:

$$\texttt{SET STATE s;} \rightarrow \begin{cases} \texttt{ProcStates[i].cur\_state = j;} \\ \texttt{ProcStates[i].T = 0;} \end{cases}$$

$$\texttt{RESTART;} \rightarrow \begin{cases} \texttt{ProcStates[i].cur\_state = 0;} \\ \texttt{ProcStates[i].T = 0;} \end{cases}$$

$$\texttt{START PROC P;} \rightarrow \begin{cases} \texttt{ProcStates[k].cur\_state = 0;} \\ \texttt{ProcStates[k].T = 0;} \end{cases}$$

$$\texttt{STOP PROC P;} \rightarrow \texttt{ProcStates[k].cur\_state = STATE\_OF\_STOP;}$$

$$\texttt{STOP;} \rightarrow \texttt{ProcStates[i].cur\_state = STATE\_OF\_STOP;}$$

$$\texttt{ERROR;} \rightarrow \texttt{ProcStates[i].cur\_state = STATE\_OF\_ERROR;}$$

$$\texttt{RESET TIMEOUT;} \rightarrow \texttt{ProcStates[i].T = 0;}$$

$$\texttt{TIMEOUT N <RS>} \rightarrow \texttt{IF(ProcStates[i].T > N) <CS>}$$

$$\texttt{PROC P IN STATE ACTIVE} \Leftrightarrow \begin{cases} \texttt{ProcStates[i].cur\_state != STATE\_OF\_STOP \&\&} \\ \texttt{ProcStates[i].cur\_state != STATE\_OF\_ERROR} \end{cases}$$

$$\texttt{PROC P IN STATE PASSIVE} \Leftrightarrow \begin{cases} \texttt{ProcStates[i].cur\_state == STATE\_OF\_STOP ||} \\ \texttt{ProcStates[i].cur\_state == STATE\_OF\_ERROR} \end{cases}$$

Variables and Ports So far we have defined semantics for most syntax constructs that are new in Reflex, i.e. not present in C. One important point that has been left out so far, is how Reflex treats variables that are associated with I/O ports. In our hand dryer example we have two variables associated with I/O ports. The input variable `S_HANDS_UNDER_DRYER` contains the binary value from the IR sensor and the output variable `C_TURN_ON_DRYER` is written by the program to control the joint fan/heater actuator. While all internal variables are treated in Reflex the same way as they would be in C, variables bound to ports have a somewhat different semantics.

For input variables, their value is read from the corresponding bit of a corresponding port in the Input function at the start of hyperprocess execution (see Listing 4) and cannot be modified during the hyperprocess execution.

As for the output variables, their values are read in the Input function as well, but are stored in a double-buffered manner, i.e. each value is stored in two instances, both used when evaluating and executing expressions. One instance is only used in read operations and maintains its initial value throughout the hyperprocess execution, so that each process reads the same value, regardless of their order. The second instance stores the modified value, used in write operations. This can be seen in Listing 5: variable `C_TURN_ON_DRYER` is translated into an array with two values `char POV1[2]`, and only the second value `POV1[1]` is used in write operations. At the end of hyperprocess execution, the Output function is called, that checks what output values have been modified and stores those as corresponding bits in the output ports.

6. Practical Example

The Reflex language has passed strong examination in several complex projects. One such application was in an automated control system for a single crystal growth furnace [27]. The task involved typical industrial environment and equipment with distributed functioning, intelligent sensors, four-coordinate motion subsystem, actuators, gas-vacuum subsystem with doubled pumps, cooling subsystem, heater subsystem, conventional logic and analog I/O modules, complex algorithm with active reaction on faults. A Micro PC with a CPU685E, 300 MHz [28] processor has been used as computational platform.

Рис. 1: Single Crystal Growth Furnace

The control system has been implemented with all control logic written in Reflex. The following results have been obtained:

- the control algorithm has naturally fit the hyperprocess model with a surprisingly large quantity of concurrent processes (more than 760),
- low time consumption for process execution has been logged (2 microseconds per process),
- at peak load, full cycle of hyperprocess execution took 8 milliseconds (about 10 microseconds a process),
- the Reflex language showed to be easily understood by project members who were previously not familiar with programming,
- locality of change was preserved throughout the whole development of the system,

Overall, the language has received favorable comments from programmers, and what is more, very good responses have been received from control system developers with strong background in the IEC 61131-3 standard. The language has been praised for being more flexible and comfortable as compared to the IEC languages. The flexibility of the approach and Reflex language was recently used in the project on dynamic verification of control programs [29].

7. Conclusion

In this work we have presented a novel programming language Reflex that provides a practical notation for digital control systems. The hyperprocess model underlying Reflex takes is compliant with primary features of control software – cyclic execution, event-driven and polymorphic behavior, concurrency, etc. The language syntax is similar to C and provides time-interval operations as well as means for inter-process communication that encourage hierarchical organization of control software. A translator from Reflex to C has been implemented and semantics of key Reflex constructs has been demonstrated based on this translation.

The language has been successfully used in multiple industrial applications and demonstrated promising qualities, producing easily readable, maintainable code and overall robust and dependable software.

Further direction of work would be to research applicability of verification methods to software specified in Reflex.

References

1. Hoare C. A. R. Communicating Sequential Processes. Prentice Hall, 1985. – 256 p.
2. Harel D. Statecharts: a Visual Formalism for Complex Systems. / In: Science of Computer Programming 8. Elsevier Science Publishers B. V., North-Holland. 1987. P. 231–274.
3. Lynch N., Tuttle M. An Introduction to Input/Output Automata // CWI Quarterly. 1989. No 2. P. 219–246.
4. Berry G. The Foundations of Esterel // In: Plotkin, G., Stirling, C., and Tofte, M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press, Foundations of Computing Series. 2000. P. 425–454.
5. Milner R. Communication and Concurrency. Series in Computer Science. Prentice Hall, 1989. – 300 p.
6. Shoshmina I. V. Developing formal temporal requirements to distributed program systems // System Informatics. 2016. No. 8 P. 33–42.
7. Davis J., Schneider S. An Introduction to Timed CSP // PRG-75, PRG Programming Research Group Oxford. 1989.
8. Chen L., Anderson S., and Moller F. A Timed Calculus of Communicating Systems // Technical Report LFCS-90-127, University of Edinburgh. 1990.
9. Kaynar D. K., Lynch N., Segala R., Vaandrager F. Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems // 24th IEEE International Real-Time Systems Symposium (RTSS'03), IEEE Computer Society Cancun, Mexico. 2003). P. 166–177.
10. Staroletov S. Design and Implementation a Software for Water Purification with Using Automata

Approach and Specification Based Analysis // System Informatics. 2017. No10. P. 33–44.

11. Shabaldina N., Gromov M. Using BALM-II for deriving parallel composition of timed finite state machines with outputs delays and timeouts: work-in-progress // System Informatics. 2016. No. 8 P. 33–42.
12. Abadi M., Lamport L. An Old-fashioned Recipe for Real Time // ACM Transactions on Programming Languages and Systems, Vol. 16 (5), ACM Press New York, Sept 1995. P. 1543–1571.
13. IEC 61131-3: Programmable controllers Part 3: Programming languages, International Electrotechnical Commission Std., Rev. 2.0, 2003.
14. Basile F., Chiacchio P., and Gerbasio D. On the Implementation of Industrial Automation Systems Based on PLC // IEEE Trans. on Automation Science and Engineering, Oct 2013, vol. 10, no. 4, pp. 990–1003.
15. Crater K. C. When Technology Standards Become Counterproductive. Control Technology Corporation. 1992.
16. Thramboulidis K. and Frey G. An MDD Process for IEC 61131-based Industrial Automation Systems // in 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA11), September 5-9, 2011, Toulouse, France, 2011. P. 1–8
17. IEC 61499: Function Blocks for Industrial Process Measurement andControl Systems, Parts 1 – 4, International Electrotechnical Commission Std., Rev. 1.0, 2004/2005.
18. Wagner F., Schmuki R., Wagner T., and Wolstenholme P. Modeling Software with Finite State Machines.Boston, MA, USA: Auerbach Publications, 2006.
19. Malyshkin V.E. Parallel computing technologies 2016 // The Journal of Supercomputing, Vol. 73, Iss. 2, Springer, 2017. P. 607–608.
20. Zyubin V.E. Multicore Processors and Programming. Open Systems J., N7-8. 2005. P. 12–19 (in Russian).
21. Zyubin V.E. Text and Graphics: What Language Does Programmer Need? // Open Systems J., 2004. No 1. P. 54–58 (in Russian).
22. Glushkov V. M. The Synthesis of Digital Automata. PhisMathGis, Moscow. 1962 (in Russian)
23. Wagner F., Wolstenholme P.: Misunderstandings about State Machines // IEE J. Computing and Control Engineering, Vol. 16. No 4, Aug 2004. P. 45.
24. Zyubin V. E. Hyper-automaton: A Model of Control Algorithms // in IEEE International Siberian Conference on Control and Communications (SIBCON-2007). Proceedings of the IEEE International Siberian Conference on Control and Communications, O. Stukach, Ed. Tomsk, Russia: IEEE, 2007, pp. 51–57. Available: https://doi.org/10.1109/SIBCON.2007.371297.
25. Kof L., Schätz B. Combining Aspects of Reactive Systems // In: Proc. of Andrei Ershov Fifth Int. Conf. Perspectives of System Informatics. Novosibirsk. 2003. P. 239–243.
26. Slonneger K. and Kurtz B. L. Formal syntax and semantics of programming languages / Addison-Wesley Reading, 1995, – 340 p.
27. Lubkov A. A., Zyubin V. E., Kurochkin A. V., Budnikov K. I. A Control System Architecture for a Single Crystal Growing Furnace.// In: Proc. of the Second IASTED Int. Conf. Automation,

Control, and Applications. 2005. P. 166–169.

28. CPU686E. CPU Card. Technical Specifications. Fastwel Inc. 2005.
29. Liakh T., Zyubin V. Model Checking of Industrial Control Algorithms in Combination with Virtual Objects // System Informatics. 2016. No 8. P. 11–20. (In Russian).

A. Appendix A. Reflex formal syntax in EBNF

```
program =   "PROGR", id, "{", [tact], [{const_or_enum_spec}],
            [{function_decl}], [{register_spec}], {process_spec},
            "}";

tact  =   "TACT", int_num, ";";

const_or_enum_spec  = const_spec | enumerator_spec;
const_spec          = "CONST", const_id, const_exp_body, ";";
enumerator_spec     = "ENUM", "{", enumerator_list, "}";
enumerator_list     = enumerator | (enumerator, ",", enumerator_list);
enumerator          = const_id | (const_id, "=", const_exp_body);
const_exp_body      = const_pref_term |
                      (const_pref_term, {const_infix, const_pref_term});
const_pref_term     = [const_prefix], const_term;
const_prefix        = "~" | "!" | "+" | "-";
const_infix         = "+" | "-" | "*" | "/" | "%" | "<<" | ">>" | "&" |
                      "^" | "|" | "&&" | "||";
const_term          = number | const_id | "(", const_exp_body,")"  ;
const_id            = id;

function_decl = c_type_spec, func_id, "(", c_type_spec_list, ")", ";";
c_type_spec_list  = c_type_spec | (c_type_spec, ",", c_type_spec_list);
func_id = id;

register_spec = ("INPUT" | "OUTPUT"), reg_id, addr_1, addr_2,
                register_size;
addr_1        = int_num;
addr_2        = int_num;
register_size = "8" | "16";
reg_id        = id;

process_spec    = "PROC", proc_id, "{", [var_list], {func_state}, "}";
proc_id         = id;
var_list        = {var_spec | var_decl};
var_spec        = (phys_var_spec | calc_var_spec),
                  visibility_spec, ";";
phys_var_spec   = int_type_spec, var_id, "=",
                  "{", reg_bits_spec_list, "}";
reg_bits_spec_list  = reg_bits_spec |
                      (reg_bits_spec, ",", reg_bits_spec_list);
reg_bits_spec   = reg_id, "[", int_num, "]";
calc_var_spec   = (c_type_spec | "BOOL"), var_id;
visibility_spec = "LOCAL" | ("FOR", "ALL") | ("FOR", proc_id_list);
proc_id_list    = proc_id | (proc_id, "," , proc_id_list);

var_decl        = "FROM", "PROC", proc_id, var_id_list, ";";
var_id_list     = var_id | (var_id, ",", var_id_list);


func_state      = "STATE", func_state_id, "{",
                  ([func_state_body],timeout_react_spec) |
```

```
                  timeout_react_spec, "}";
func_state_body = {react_spec};
react_spec      = ";" | ("{", func_state_body, "}") | switch_spec |
                  event_react_spec | start_spec | stop_spec |
                  error_spec | loop_decl | set_cur_sf_spec |
                  restart_cur_proc_spec | reset_timer_spec |
                  var_equation;

switch_spec     = "SWITCH", "(", event, ")", "{", {case_spec}, "}";
case_spec       = "CASE", int_num, ":", func_state_body,
                  ["BREAK", ";"];

event_react_spec  = event_react_body, [rev_event_react_body];
event_react_body  = "IF", "(", event, ")", react_spec;

rev_event_react_body= "ELSE", react_spec;

start_spec      = "START", proc_id, ";";
stop_spec       = "STOP", [proc_id], ";";
error_spec      = "ERROR", [proc_id], ";";
loop_decl       = "LOOP", ";";
set_cur_sf_spec = "SET", ("STATE", func_state_id) | "NEXT", ";";
restart_cur_proc_spec = "RESTART", ";";
reset_timer_spec      = "RESET", "TIMEOUT", ";";

var_equation    = var_id, "=", event, ";");

event = var_pref_post_term |
        var_pref_post_term, {var_infix, var_pref_post_term};
var_pref_post_term  = var_prefix, term, [var_postfix];
var_prefix    = [("~" | "!" | "++" | "--" | "+" | "-" | "*" | "&")],
                ["(", c_type_spec,")"];
var_postfix   = "++" | "--";
var_infix     = "+" | "-" | "*" | "/" | "%" | "<<" | ">>" | "&" | "^" |
                "|" | "&&" | "||" | "=" | "*=" | "/=" | "%=" | "+=" |
                "-=" | "<<=", | ">>=" | "&=" | "^=" | "|=" | "<" | ">"|
                "<=" | ">=" | "==" | "!=";
term  = number | const_id | var_id | function |
        situation | "(", event, ")";
function  = func_id, "(", func_param_list, ")";
func_param_list = event | (event, {",", event});

situation   = "PROC", proc_id, "IN", "STATE",
              (func_state_id | "ACTIVE" | "PASSIVE");

timeout_react_spec  = "TIMEOUT", (number | const_id | var_id),
                      react_spec;
func_state_id = id;

var_id  = id;

c_type_spec     = "VOID" | "FLOAT" | "DOUBLE" |
                  (["SIGNED" | "UNSIGNED"],
                  ("SHORT" | "INT" | "LONG"));
int_type_spec   = "BOOL" | "SHORT" | "INT" | "LONG";
float_type_spec = "FLOAT" | "DOUBLE";

id  = letter, {letter | decimal_digit};

number      = int_num | float_num;
int_num     = octal_num | decimal_num | hex_num;

letter        = ("A" ... "Z") | ("a" ... "z") | "_";
```

```
octal_num     = "0", [{octal_digit}], ["U" | "u"] | ["L" | "l"];
octal_digit   = "0" ... "7";
decimal_num   = (decimal_digit - "0"), [{decimal_digit}], ["U" | "u"] |
                ["L" | "l"];
decimal_digit = octal_digit | "8" | "9";
hex_num       = hex_prefix, {hex_digit}, ["U" | "u"] | ["L" | "l"];
hex_digit     = decimal_digit | ("A" ... "F") | ("a" ... "f");
float_num     = decimal_float_num | hex_float_num;

decimal_float_num = (fractional_part, [exponent_part], [float_suffix])|
                    (decimal_sequence, exponent_part, [float_suffix]);
hex_float_num     = hex_prefix, (hex_fractional_part | hex_sequence),
                    bin_exponent_part, [float_suffix];

hex_prefix  = "0x" | "0X";

fractional_part = ([decimal_sequence], ".", decimal_sequence) |
                  (decimal_sequence, ".");
exponent_part   = ("e" | "E"), [sign], decimal_sequence;
sign            = "+" | "-";

hex_fractional_part = ([hex_sequence], ".", hex_sequence) |
                      (hex_sequence, ".");

bin_exponent_part = ("p" | "P"), [sign], decimal_sequence;

hex_sequence        = {hex_digit};
decimal_sequence    = {decimal_digit};

float_suffix    = "f" | "F" | "l" | "L";
```

Listing 6: Reflex Grammar Specification