Towards safe cyber-physical systems: the Reflex language and its transformational semantics

Igor Anureev
and Natalia Garanina
A.P. Ershov Institute
of Informatics Systems
Novosibirsrk, Russia
Email: anureev@gmail.com

Tatiana Liakh, Andrei Rozov
and Vladimir Zyubin
IAE SB RAS/NSU
Novosibirsk, Russia
Email: zyubin@iae.nsk.su

***Abstract*—Reflex is a process-oriented language that provides design of easy-to-maintain control software. The language has been successfully used in several safety-critical cyber-physical systems, e. g. control software for a silicon single crystal growth furnace. Now, the main goal of the Reflex language project is development a support for computer aided software engineering targeted to safety-critical application. The current issue of the project we discuss in this paper is creating static verification methods for Reflex programs. As base of the most static verification techniques is a formal language semantics, this paper presents the Reflex language semantics in form of the transformational one.**

1. Introduction

The increasing complexity and use of embedded and cyber physical systems in our lives requires a reassessment of the design and development tools. Most challenging are safety-critical systems, where incorrect behavior and/or lack of robustness lead to unacceptable loss in funds or even human life. Such systems are widely spread in industry, especially, in chemistry and metallurgy plants. Since behavior of cyber-physical system is determined by the control system, and behaviour of control system is specified by software, the study of control software is of the great interest. The correct behavior under various environmental conditions must be ensured. In case of a hardware failure, e. g. plant damage or actuator fault, the control system must automatically react to prevent dangerous consequences. This is commonly referred to as fault tolerant behavior [1]. Because of the domain specificity control systems are based on industrial controllers (PLCs) and specialized languages are used for control software design.

Industrial controllers are inherently open (i. e. communicate with an external environment), reactive (have event-driven behaviour) and concurrent (have to process a multiple asynchronous events). These features lead to special languages being used in development of control software, e.g. the IEC 61131-3 languages [2] which are the most popular in the PLC domain. However, as the complexity of control software increases and quality is of higher priority, the 35 years old technology based on the IEC 61131-3 approach is not able to address the present-day requirements [3]. This motivates researchers to enrich the IEC 61131-3 development model with object-oriented concepts [4], or develop alternative approaches, e. g. [5], [6], [7], [8].

To address the restrictions and challenges in development of present-day complex control software, the process-oriented programming (POP) has been suggested in [9]. POP involves expressing control software as a set of interacting processes, where processes are finite state automata enhanced with inactive states as well as special operators that implement concurrent control flows and time-interval handling. Comparing to well-known FSA modifications, e. g. Communicating Sequential Processes [10], Harels Statecharts [11], Input / Output Automata [12], Esterel [13], Hybrid Automata [17], Calculus of Communicating Systems [14], and their timed extensions [15], [16], the technique both provides means to specify concurrency and saves the linearity of the control flow at the processes level. Therefore it provides a conceptual framework for process-oriented languages that are suitable to design software for cyber-physical systems.

The process-oriented approach has been implemented in domain-specific programming languages such as SPARM [18], Reflex [19] and IndustrialC [20]. 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.

The SPARM language is a predecessor of the Reflex language and now it is out of use. IndustrialC targets strict utilization of microcontroller peripheral (registers, timers, PWM, etc.) and extends Reflex with means for interrupt handling. While Reflex is a pure process-oriented language that assumes strict encapsulation of platform-dependent I/O subroutines into a library. As well as it is done in the IEC 61131-3 languages. This encapsulation provides semantic simplicity of the language that, together with the continuing practical value, makes it very attractive for theoretical

studies.

Application domain for Reflex language includes various kinds of control algorithms, PLC-based control systems, hybrid and cyber-physical systems. A Reflex program is specified as a set of communicating concurrent processes. Specialized constructs have been introduced for controlling of processes and time intervals handling. Reflex also provides constructs for linking its variables to physical I/O signals. Procedures for reading / writing data through registers and their mapping to variables are generated automatically by the translator.

Reflex has been successfully used in several safety-critical cyber-physical systems, e. g. control software for a silicon single crystal growth furnace [21]. Currently Reflex project is focused on design and development tools for safety-critical systems. Because of its system independence Reflex easily integrates with LabVIEW [22]. This allows to develop software combining event-driven behavior with advanced graphic user interface, remote sensors and actuators, LabVIEW-supported devices, etc. Using flexibility of LabVIEW, a set of plant simulators was designed for the learning purposes [23]. The LabVIEW-based simulators include 2D animation, tools for debugging, and language support for learning of control software design. One of the result obtained in this direction is LabVIEW-based dynamic verification toolset for Reflex programs.

Dynamic verification treats the software as a black-box, and checks its compliance with the requirements by observing run-time behavior of the software under a set of test-cases. While such a procedure can help detect the presence of bugs in the software, it cannot guarantee their absence [24].

Unlike dynamic verification, static methods are based on source code analysis and are commonly recognized as the only way to ensure required properties of the software. It is therefore very important to adopt static verification methods for Reflex programs.

Static verification methods require programs to have formal semantics. There are three basic approaches to formal semantics of programming languages: operational, axiomatic and denotational. Operational semantics describes the execution of programming language constructs in terms of states and transitions from one state to other. Axiomatic semantics gives meaning to language construct by logical formulas. Denotational semantics interprets each language construct as a denotation. These denotations are usually described in a denotational metalanguage. If a denotational semantics uses the source language as a denotational metalanguage, it is called a transformational semantics.

Developing formal (operational, axiomatic or denotational) semantics of Reflex from scratch would be a very complex and time-consuming task, since it would require to formalize all constructs of the C language for which Reflex is an extension.

Instead, we define the transfomational semantics of Reflex using C as a denotational metalanguage and, thus, reduce the task of development of Reflex semantics to the task of development of C semantics for which there are several solutions based on operational [25], [26], axiomatic [27], [28] and denotational [33], [34], [35] approaches.

2. Introduction to Reflex

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 has been specified in [19].

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. Trivial at first sight, the task is complicated with discontinuity of the input signal caused by the user rubbing and turning their hands under the dryer. To avoid erratic toggling of the dryer heater and fan, the program should not react to brief interruptions in the signal and the actuators should only be turned off once the sensor reading is a steady ”off”. The control algorithm can only meet this requirement by measuring the duration of the off state of the sensor. In this case, a continuous ”off” signal longer than a certain given time (for example, 1 s) would be regarded as a ”hands removed” event.

```
PROGR HandDryerController {
  TACT 100;
  CONST ON 1;
  CONST OFF 0;
/*==============================*/
/* I/O ports specification      */
/* direction, name, address,    */
/* offset, size of the port     */
/*==============================*/
  INPUT  SENSOR_PORT  0 0 8;
  OUTPUT ACTUATOR_PORT 1 0 8;

/*==============================*/
/* processes definition         */
/*==============================*/
  PROC Init {
/*===== VARIABLES =============*/
    BOOL I_HANDS =
      {SENSOR_PORT[1]} FOR ALL;
    BOOL O_DRYER =
      {ACTUATOR_PORT[1]} FOR ALL;

/*===== STATES ================*/
    STATE Waiting {
      IF (I_HANDS == ON) {
        O_DRYER = ON;
        SET NEXT;
      } ELSE O_DRYER = OFF;
    }
    STATE Drying {
      IF (I_HANDS == ON)
        RESET TIMEOUT;
        TIMEOUT 10
          SET STATE Waiting;
    }
  } /* \PROC */
}  /* \PROGRAM */
```

Listing 1. Hand dryer example in Reflex

In Reflex, a program is presented as a set of concurrently running communicating processes, each defined in textual form starting with a **PROC** keyword:

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

The first process defined in the text is initially active when the program is started.

Program execution is split into clocks with a fixed period specified with the **TACT** directive at the top of the code.

The body of a process consists variable declarations and list of state function definitions in the following form:

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

The state that is defined first in the process body is one into which that process is transitioned by **START PROC** statements. Two extra states **STOP** and **ERROR** are defined implicitly for each process.

The body of a state is defined as a sequential block of code, consisting of the assignment statements, if statements, switch statements, process control statements and one optional timeout statement that define events and their corresponding reactions. To prevent the code from blocking the program execution, Reflex does not provide any loop statements.

The syntax for expression and selection statements is almost identical to that in C the selection statements is very similar to that of equivalent C statements and is discussed in detail in [19]. For introduction purposes here we focus on those constructs that are specific to Reflex.

Process control and communication in Reflex is managed using state transitions, control statements and activity predicates that can be used in expressions. State can be only be used by the process on itself and 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 program 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;
```

Processes are also able to check whether other processes are in their active or passive states using selection statements in conjunction with **ACTIVE/PASSIVE** predicates, e.g.:

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

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

```
TIMEOUT <clocks num> <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.

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>;
```

One important feature of variables bound to ports is that all read and write operations for these variables are double-buffered. The values of I/O ports are read once per program cycle and each value is stored in two instances – one for read and one for write operations. New values for the output ports are set and sent to external devices at the end of the cycle. This way all processes read the same port values even if they are modified inside that cycle of execution.

3. Reflex Semantics

Transformational semantics of Reflex has the following restrictions:

- Transformational semantics is defined only for well-formed programs.
- Information about ports and matching variables with ports is not taken into account as it relates to communication with physical devices.
- Variable access levels are not taken into account, since they determine only the correct access to variables, which is provided by well-formed programs.
- We consider that processes are executed sequentially in each tact.

Let C_R and C_C be sets of constructs of programs in Reflex and C, respectively. Programs in these languages are also included in these sets. Let $C = C_R \cup C_C$. Transformational semantics of Reflex is given by the binary relation $\rightsquigarrow \in C \times C$ such that $\neg(c_1 \rightsquigarrow c_2)$ for $c_1, c_2 \in C_C$.

Let p_R be a Reflex program that is transformed into C program p_C, i. e. $p_R \rightsquigarrow p_C$. Let $P = \{p_1, \ldots, p_n\}$ and $S = \{s_1, \ldots, s_m\}$ be sets of names (identifiers) of processes and process states of p_R, respectively. Let e and ss be an expression and a statement sequence in Reflex, respectively.

The transformational semantics of Reflex is defined by the following transformation rules for its constructs.

Programs. Let dec_T be a tact declaration in p_R. Let dec_v and dec_c be lists of all variable and constant declarations in p_R, respectively. Let m_i be the number of states of p_i, and $s^i_1, \ldots, s^i_{m_i} \in S$ be states of p_i for $1 \leq i \leq n$.

To preserve information about input ports (more exactly about changing the values of variables of p_R through these

ports) the set of variables of p_R is divided into three pair-disjoint subsets: externally initialized variables (that get the value once directly after initialization of processes of p_R), externally changed variables (their values can be externally changed before each tact) and externally unchanged variables (their values cannot be externally changed).

Let $\{v^i_1, \ldots, v^i_{k_i}\}$ and $\{v^e_1, \ldots, v^e_{k_e}\}$ be sets of externally initialized and externally changed variables of the types $t^i_1, \ldots, t^i_{k_i}$ and $t^e_1, \ldots, t^e_{k_e}$ in p_R, respectively. The relation $\rightsquigarrow$ implicitly depends on these sets. The rule for p_R has the form:

$p_R \rightsquigarrow$
dec_T
dec_c
enum $states$ {s_{stop}, s_{error},
s^1_1, ..., $s^1_{m_1}$, s^2_1, ..., $s^2_{m_2}$, ..., s^n_1, ..., $s^n_{m_n}$};
states cs_1, ..., cs_n;
long $clock_1$, ..., $clock_n$;
dec_v
$init$();
for (;;) {
v^e_1 = $input_{t^e_1}$(); ... $v^e_{k_i}$ = $input_{t^e_{k_i}}$ ();
$exec_1$(); ... $exec_n$();
$clock_1$++; ... $clock_n$++;}.

The enumeration type $states$ defines stop state, error state and states of processes in p_R, respectively. The values of variables $cs_1, \ldots, cs_n$ are current states of processes $p_1, \ldots, p_n$, respectively. The values of variables $clock_1, \ldots, clock_n$ (called clock variables) are current time (measured in tacts) of execution of processes $p_1, \ldots, p_n$, respectively, in their last current states.

For port declarations are not included in the right part of the rule, they are eliminated from p_R. The function $input_t$ (called an input function) with the prototype

t $input_t$(void);

returns an arbitrary value of the Reflex type t. Its concrete implementation is not important for the purposes of deductive verification. The family of such functions (with the index t) models external changes of the values of variables of p_R.

The function $init$ initializes the processes of p_R before their first launch:

void $init$() {
cs_1 = s^1_1; cs_2 = s_{stop}; ... cs_n = s_{stop};
$clock_1$ = 0; ... $clock_n$ = 0;
v^i_1 = $input_{t^i_1}$(); ... $v^i_{k_i}$ = $input_{t^i_{k_i}}$ ();}.

Let $body(s_{ij})$ denote the body of definition of state s_{ij} of process p_i from which all variable declarations are eliminated. The function $exec_i$ defines execution of p_i:

void $exec_i$() {
switch (cs_i) {
case s^i_{i1}: $body(s_{i1})$ break;
...
case $s^i_{ik_i}$: $body(s_{ik(i)})$ break;}}.

The substitution rule. Since Reflex is an extension of C, constructs in these languages in Reflex program can be embedded into each other. Transformational semantics of such embedding is given by the substitution rule.

Let $c, c', c_1, c_2 \in C$, and $c[c']$ denote the place of the occurrence of c' into c. The substitution rule has the form:

If $c_1 \rightsquigarrow c_2$, then $c[c_1] \rightsquigarrow c[c_2]$.

Types. Type $bool$ is defined by the rule:

$bool \rightsquigarrow \_Bool$.

The rest Reflex types are C types.

The tact declaration. Let n be a number. The tact declaration is defined by the rule:

tact n; $\rightsquigarrow$ #define $tact$ n.

Constant declarations. Let id be a constant name. Constant declarations are defined by the rules:

const id n; $\rightsquigarrow$ #define id n
const id e; $\rightsquigarrow$ #define id (e)
enum bod $\rightsquigarrow$ enum t bod;.

Here t is a new enumeration type with the body bod.

Variable declarations. Let ϵ denote an empty string. Variable declaration are defined by the rules:

t id ...; $\rightsquigarrow$ t id;
from proc pj id; $\rightsquigarrow$ ϵ.

Thus, information about matching variables with ports and variable access levels is deleted, and Reflex variable declaration is transformed into a C variable declaration.

The below statements are supposed to be found in p_i.

State operations. Transformational semantics of state operations is defined by the rules:

(is active) $\rightsquigarrow$ (p_i is active);
(p_j is active) $\rightsquigarrow$ ((cs_j != s_{stop}) && (cs_j != s_{error}));
(is inactive) $\rightsquigarrow$ (p_i is inactive);
(p_j is inactive) $\rightsquigarrow$ ((cs_j == s_{stop}) || (cs_j == s_{error}));
(in state stop) $\rightsquigarrow$ (p_i in state stop);
(p_j in state stop) $\rightsquigarrow$ (cs_j == s_{stop});
(in state error) $\rightsquigarrow$ (p_i in state error);
(p_j in state error) $\rightsquigarrow$ (cs_j == s_{error}).

Process control statements. Process control statements include stop statements, error statements, start statements, set statements, next statements, and restart statements.

The stop statement is defined by the rules:
stop; $\rightsquigarrow$ stop proc p_i;
stop proc p_j; $\rightsquigarrow$ {$clock_j$ = 0; cs_j = s_{stop};}.
The error statement is defined by the rules:
error; $\rightsquigarrow$ error proc p_i;
error proc p_j; $\rightsquigarrow$ {$clock_j$ = 0; cs_j = s_{error};}.
The start statement is defined by the rule:
start proc p_j; $\rightsquigarrow$ {$clock_j$ = 0; cs_j = s^j_1;}.
The set statement is defined by the rule:
set state s^i_j; $\rightsquigarrow$ {$clock_i$ = 0; cs_i = s^i_j;}.
The next statement is defined by the rule:
If $cs_i = s^i_j$, then next; $\rightsquigarrow$ {$clock_i$ = 0; cs_i = s^i_{j+1};}.
This statement can not occur in the body of the last state declaration of p_i since transformational semantics is defined only for well-formed programs.
The restart statement is defined by the rule:
restart; $\rightsquigarrow$ {$clock_i$ = 0; cs_i = s^i_1;}.

Timeout control statements. Process control statements include the reset statement and the timeout statement.

The reset statement is defined by the rule:

reset timeout; $\rightsquigarrow clock_i$ = 0;.

The timeout statement is defined by the rule:

timeout e ss $\rightsquigarrow$ if ($clock_i >= e$) ss.

C code insertions. C code can be inserted into Reflex program as the lines starting with two symbols "#" and "C". Transformational semantics of such insertions is defined by the rule:

#C $c_C \rightsquigarrow c_C$.

4. The transformation example

The result of transformation of the program controlling a hand dryer considered in section 2 has the form:

```
#define TACT 100
#define ON 1
#define OFF 0

enum states {stop_state, error_state,
 Init_Waiting, Init_Drying};
states Init_state;
long Init_clock;

_Bool I_HANDS;
_Bool O_DRYER;

void init() {
 Init_state = Init_Waiting;
 Init_clock = 0;}

void Init_exec() {
 switch (Init_state) {
  case Init_Waiting:
   if(I_HANDS == ON)
    {O_DRYER = ON;
     Init_clock = 0;
     Init_state = Init_Drying;}
   else O_DRYER = OFF;
  case Init_Drying:
   if(I_HANDS == ON) {
     Init_clock = 0;
     Init_ state = Init_ Drying;}
   if(Init_clock >= 10) {
     Init_clock = 0;
     Init_ state = Init_Waiting;}}}

void main(void){
 init();
 for(;;) {I_HANDS = input_bool();
  Init_exec(); Init_clock++;}}
```

Listing 2. Hand dryer case study generated C code

The stop and error states are denoted by `stop_state` and `error_state`. The process states are denoted by concatenations of process names, the symbol _ and state names, for example `Init_Waiting`. The current state variables are denoted by concatenations of process names and the string `_state`, for example `Init_state`. The clock variables are denoted by concatenations of process names and the string `_clock`, for example `Init_clock`. The input functions are denoted by concatenations of the string `input_` and the corresponding types, for example `input_bool`.

5. Discussion and Conclusion

The proposed transformational semantics of Reflex has several remarkable properties.

First, it is compact and intuitive.

Second, if a fragment of C has a formal (operational, axiomatic or denotational) semantics, this semantics is relatively easily transferred to Reflex extension of the fragment. Thus, we get formal (operational, axiomatic or denotational) semantics of the corresponding Reflex fragments as a bonus.

Third, the transformational semantics are invariant with respect to the safety-oriented fragments (sublanguages) of C such as C0 [29], Clight [30], C-light [31] and MISRA C [32]. This means that if all C constructs in a source Reflex program belong to a fragment of the above, then the target program in C (the result of transformations of the source program) also belongs to this fragment. Thus, the formal methods (for example, verification methods) developed for these sublanguages can be extended to the corresponding subclasses of Reflex programs.

Fourth, the proposed concepts of externally initialized, externally changed and externally unchanged variables allows to model interaction with input ports at an abstract level.

A notable feature of this approach is its seamlessness. Most static verification methods require the system to be expressed in some specialized modelling language with that expression only being used for verification purposes. Hence, designers have to create two separate specifications for their system – one for verification, and another one for generating executable code. The transformation between the two representations is performed manually and there is no guarantee that the code verified is the same code as that which is consequently compiled and executed.

In the proposed approach, a single notation – Refex – is used for both. Reflex is naturally translated into a rather limited subset of C which is also a subset of multiple C-like languages with well-defined semantics. Therefore the existing Reflex translator can easily be used, with minor modifications, to generate code that is fit for verification.

This grants two important benefits. Firstly, this ensures that the semantics of the verified algorithm are the same as that of the final executable program. Secondly, any modifications to the source code are automatically applied to both the model code (used for verification) and that of the target software. This way, the workload during code change is significantly reduced along with probability of human error. This is especially important for iterative development where source code is constantly modified.

In the future we plan to use this approach together with the ontological approach [36] to formal verification of concurrent systems, translating Reflex programs and their specifications into process ontology and requirements ontology, respectively.

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

References

[1] M. Blanke, M. Kinnaert, J. Lunze, and M. Staroswiecki, Diagnosis and Fault-Tolerant Control, 2nd ed. Springer-Verlag Berlin Heidelberg, 2006.

[2] IEC 61131-3: Programmable controllers Part 3: Programming languages, International Electrotechnical Commission Std., Rev. 2.0, 2003.

[3] 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.

[4] 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

[5] IEC 61499: Function Blocks for Industrial Process Measurement and-Control Systems, Parts 1 – 4, International Electrotechnical Commission Std., Rev. 1.0, 2004/2005.

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

[7] Samek, M. Practical UML statecharts in C/C++: event-driven programming for embedded systems. 2nd edition. Oxford: Newnes, 2009.

[8] Shalyto A. A., Tukkel' N. I. SWITCH Technology: An Automated Approach to Developing Software for Reactive Systems // Programming and Computer Software. September 2001, Volume 27, Issue 5, pp. 260–276.

[9] 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.

[10] Hoare, C. A. R.: Communicating Sequential Processes. Prentice-Hall Int. (1985).

[11] Harel, D.: Statecharts: a Visual Formalism for Complex Systems. In: Science of Computer Programming 8. Elsevier Science Publishers B.V., North-Holland (1987) 231–274.

[12] Lynch, N., Tuttle, M.: An Introduction to Input/Output Automata. CWI Quarterly, 2 (1989) 219–246.

[13] 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) 425–454.

[14] Milner, R.: Communication and Concurrency. Series in Computer Science. Prentice Hall (1989).

[15] 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), 166–177.

[16] Kof, L., Schtz, B.: Combining Aspects of Reactive Systems. In: Proc. of Andrei Ershov Fifth Int. Conf. Perspectives of System Informatics. Novosibirsk (2003) 239–243.

[17] Henzinger T.A. The Theory of Hybrid Automata. In: Inan M.K., Kurshan R.P. (eds) Verification of Digital and Hybrid Systems. NATO ASI Series (Series F: Computer and Systems Sciences), Springer, Berlin, Heidelberg. 2000. vol 170. pp. 265–292.

[18] Zyubin V. SPARM Language as a Means for Programming Microcontrollers // Optoelectronics, Instrumentation, and Data Processing, 1996, v. 2, pp. 36–44.

[19] Liakh T. V., Rozov A. S., Zyubin V. E. *Reflex Language: a Practical Notation for Cyber-Physical Systems*, System Informatics, No. 12, 2018, pp. 85–104.

[20] Andrei S. Rozov, Vladimir E. Zyubin Process-oriented programming language for MCU-based automation // Proceedings of the IEEE International Siberian Conference on Control and Communications, Tomsk, Russia, 2013, pp. 1–4.

[21] Bulavskij D., Zyubin V., Karlson N., Krivoruchko V., Mironov V. An Automated Control System for a Silicon Single-Crystal Growth Furnace // Optoelectronics, instrumentation, and data processing. 1996, v. 2, pp. 25 - 30.

[22] Travis J., Kring J. LabVIEW for Everyone: Graphical Programming Made Easy and Fun (3rd Edition) (National Instruments Virtual Instrumentation Series). Prentice Hall PTR, Upper Saddle River, NJ, USA. 2006.

[23] Zyubin V. Using Process-Oriented Programming in LabVIEW // Proceedings of the Second IASTED International Multi-Conference on Automation, control, and information technology: Control, Diagnostics, and Automation, Novosibirsk, June 15-18, 2010. P. 35–41.

[24] Software Engineering Techniques / Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 2731 October 1969 J.N. Buxton and B. Randell, eds, April 1970, p. 16.

[25] M. Norrish, *C formalised in HOL*, Ph.D. thesis, University of Cambridge, Technical report, UCAM-CL-TR-453, 1998.

[26] Y. Gurevich, J. Huggins, *The semantics of the C programming language*, Lecture Notes in Computer Science, 702, 1993, 274–308.

[27] V. Nepomniaschy, I Anureev, A Promsky, *Towards verification of C programs: Axiomatic semantics of the C-kernel language*, Programming and Computer Science, 29(6), 2003, 338–350.

[28] I. Anureev, I. Maryasov, V. Nepomniaschy, *C-programs verification based on mixed axiomatic semantics*, Automatic Control and Computer Sciences, 45(7), 2011, 485–500.

[29] W. Paul, et al., *The Verisoft project*, http://www.verisoft.de/, 2003-2008.

[30] S. Blazy and X. Leroy, *Mechanized semantics for the Clight subset of the C language*, J. Autom. Reasoning, 43(3), 2009, 263–288.

[31] V. Nepomniaschy, I Anureev, I. Mikhailov, A Promsky, *Towards verification of C programs. C-light language and its formal semantics*, Programming and Computer Science, 28(6), 2002, 314–323.

[32] L. Hatton, *Safer language subsets: an overview and a case history, MISRA C*, Information & Software Technology, 46(7), 2004, 465–472.

[33] H. Tews, T. Weber, M. Völp, *A formal model of memory peculiarities for the verification of low-level operating-system code*, Electronic Notes in Computer Science, 217, 2008, 79–96.

[34] E. Gimenez, E. Ledinot, *Semantics of a subset of the C language*, http://coq.inria.fr/contribs/minic.html, 2004.

[35] N. Papaspyrou, *A formal semantics for the C programming language*, Ph.D. thesis, National Technical University of Athens, 1998.

[36] Natalia Garanina, Vladimir Zyubin, Tatiana Lyakh, Sergei Gorlatch. *An Ontology of Specification Patterns for Verification of Concurrent Systems.* // In: New Trends in Intelligent Software Methodologies, Tools and Techniques. Proceedings of the 17th International Conference SoMeT-18. H. Fujita and E. Herrera-Viedma (Eds.). Series: Frontiers in Artificial Intelligence and Applications, vol. 303. Amsterdam: IOS Press, 2018. P. 515528. DOI 10.3233/978-1-61499-900-3-515.