UDK 004.423.42, 004.434, 681.51

Operational Semantics of Reflex*

Anureev I.S. (A.P. Ershov Institute of Informatics Systems, Institute of Automation and Electrometry)

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. This paper presents formal operational semantics of the Reflex language as a base for applying formal methods to verification of Reflex programs.

Keywords: *operational semantics, Reflex language, control system, control software, programmable logic controller*

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.

Many control systems are based on industrial programmable logic controllers (PLCs) possessing the following features: they 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 [1] 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 [2].

*This work has been supportedby the Russian Ministry of Education and Science and the Russian Foundation for Basic Research (grants 17-07-01600 and 17-01-00789).

Reflex is a domain-specific extension of the C language developed as an alternative to IEC 61131-3 languages. A Reflex program is specified as a set of communicating concurrent processes. Specialized constructs are introduced for controlling processes and handling time intervals. Reflex also provides constructs for linking its variables to physical I/O signals. Reflex assumes scan-based execution, i.e. a time-triggered control loop, and strict encapsulation of platform-dependent I/O subroutines into a library, which is a widely applied technique in IEC 6113-3 based systems. To provide both ease of support and cross-platform portability, the generation of executable code is implemented in two stages: the Reflex translator generates C-code and then a C-compiler produces executable code for the target platform.

Currently, the Reflex project is focused on design and development tools for safety-critical systems. Because of its system independence Reflex easily integrates with LabVIEW [3]. This allows to develop software combining event-driven behavior with advanced graphic user interface, remote sensors and actuators, LabVIEW-supported devices, etc. Using the flexibility of LabVIEW, a set of plant simulators was designed for learning purposes [4]. The LabVIEW-based simulators include 2D animation, tools for debugging, and language support for learning of control software design. One of the results obtained in this direction is a 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 on a set of test-cases. While such a procedure can help detect the presence of bugs in the software, it cannot guarantee their absence [5].

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. Since these methods require programs to have formal semantics, in this paper we present formal operational semantics of Reflex programs.

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). 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. A formal Reflex syntax definition in EBNF has been specified in [6].

```
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 in milliseconds specified with the TACT directive at the top of the code.

The body of a process consists of 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 time-out 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 in Reflex is identical to that in C (except for Reflex-specific boolean operations on states called activity predicates) and is discussed in detail in [6]. 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 STOP and ERROR) 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. Operational Semantics of Reflex

Let N be a set of natural numbers (including 0). Let U^* be a set of all sequences from elements of set U, $|u|$ and $u.i$ denote length and i-th element of sequence $u \in U^*$, respectively. Let $< u_1, \ldots, u_m >$ denote a sequence of elements u_1, ..., u_m and $con(w_1, \ldots, w_n)$ denote concatenation of sequences w_1, ..., w_n.

For the set A of Reflex programs we define: sets T, C_R, H and E of types, constructs, statements and expressions, set $val(t)$ of all values of type $t \in T$, and set $\Gamma = \cup_{t \in T} val(T)$ of all values.

For each program $\alpha \in A$ we define: its environment π, set $P = \{p_1, \ldots, p_n\}$ of processes, set Θ of process states, set F of functions, set $V = V_s \cup V_l$ of variables, set $V_s = V_i \cup V_o$ of shared variables (they are shared with π), set V_l of local variables ($V_r \cap V_l = \emptyset$), set V_i of input variables (π can only write to them), set V_o of output variables (π can only read from them), $V_i \cap V_o = \emptyset$, function $vt \in V \to T$ associating variables with their types, function $pso \in P \to \Theta^*$ associating processes with ordered sequence of their states (they are listed in the order they appear in p), ordered sequence $po \in P^n$ of processes (they are listed in the order their definitions appear in α), function $psv \in \Theta \to H$ associating process states with their bodies (statements), function $par \in F \to V_l$ associating functions with their parameters, and function $bod \in F \to H$ associating functions with their bodies (statements).

For each program α we consider that the following restrictions hold:

1. Program α is well-formed.
2. Information about ports of α and matching variables with ports is not taken into account as it relates to communication with physical devices. Program α interacts with its environment directly through input and output variables.
3. Variable access levels in α are not taken into account, since they determine only the correct access to variables, which is provided for well-formed programs.
4. There is no overload of names of process states, variables and function parameters, i.

e. α is a result of renaming overloaded names. Therefore we can consider that function parameters are also variables. For each function f, new variable val_f specifying the return value of f is added to α.

5. There is no a tact declaration and constant declarations in α, i. e. α is a result of execution of all C-like macro substitutions.
6. Processes are executed sequentially in each tact.

Since Reflex is an extension of C, its operational semantics is based on a transition system $(S, S_i, \to_R, \to_C)$, where S is a set of states, $S_i \subseteq S$ is a state of initial states, $\to_R$ and $\to_C$ are transition relations for Reflex-specific constructs and C programs, respectively. There are several solutions for description of operational semantics of C [7–11]. Therefore we focus on operational semantics of Reflex-specific constructs, considering that relation $\to_C$ is determined by one of these approaches (our description is closest to the approach [10]).

Let $u.w$ denote the value of function u for argument w.

A state $s \in S$ is defined as a tuple $(gc, cp, lc, ps, vv, ih, oh, cf)$, where $gc \in N$ is a global clock, $cp \in P$ is a current process, $lc \in P \to N$ is a function associating processes with their local clocks, all clocks count time in ticks (one tick corresponds to one iteration of control loop), $ps \in P \to \Theta$ is a function associating processes with their states, $vv \in V \to \Gamma$ is a function associating variables with their values, $ih \in V_i \to \Gamma^*$ is an input history ($ih.v.i$ is a value of $v \in V_i$ written by π during i-th tick of global clock gc), $oh \in V_o \to \Gamma^*$ is an output history ($oh.v.i$ is a value of $v \in V_o$ read by π during i-th tick of global clock gc), and current function cf (its body is executed).

Let $s.gc$, ..., $s.oh$ denote access to component gc, ..., oh of state s.

A state s is initial if $gc = 0$, $lc.p = 0$ for each $p \in P$, $ps(po.1) = pso.(po.1)$, $ps.(po.i) = stop$ for each $1 < i \leq n$, $|ih.v| = 0$ for each $v \in V_i$, and $|oh.v| = 1$ for each $v \in V_o$.

For the transition system we define: set $\Xi_C = C_C \times S$ of configurations of C programs, set $\Xi_R = C_R \times S$ of configurations of Reflex programs, and set $\Xi = \Xi_C \cup \Xi_R$ of configurations. The transition relations have the following properties: $\to_C \in \Xi_C \times \Xi_C$ and $\to_R \in \Xi_R \times \Xi$.

Operational semantics of Reflex-specific constructs is defined by transition rules. Many of these rules use meta-assignment $u_1.u_2...u_m$ `:=` u`;`.

Meta-assignment. Let $upd(w, u_1.u_2. \ ... \ .u_m, u)$ be a function replacing $w.u_1.u_2. \ ... \ .u_m$ by u in w. Then meta-assignment is defined by rule

$(u_1.u_2. \ ... \ .u_m$ `:=` u`;`$, s) \to_R (u, upd(s, u_1.u_2. \ ... \ .u_m, u))$.

Program. Program α is defined by rules

$(\alpha;, s) \rightarrow_R (\texttt{CONTROL LOOP};, s)$;

$(\texttt{CONTROL LOOP};, s) \rightarrow_R$

$(\texttt{INPUT};\ p_1;\ \ \ldots\ \ p_n;\texttt{OUTPUT; TICK; CONTROL LOOP};, s)$.

Construct CONTROL LOOP; defines repeated control loop. Constructs INPUT; and OUTPUT; interact with environment, writing to input variables and reading from output variables, respectively. Construct p; executes process p. Construct TICK; increments the value of global and local clocks. These constructs are defined by rules

If $vv' \in V \rightarrow \Gamma$, and $ih'(v) = con(ih(v), < vv'(v) >)$ for each $v \in V_i$

then $(\texttt{INPUT};, s) \rightarrow_R (ih\ \texttt{:=}\ ih';, s)$;

If $oh'(v) = con(oh(v), < vv(v) >)$ for each $v \in V_o$

then $(\texttt{OUTPUT};, s) \rightarrow_R (oh\ \texttt{:=}\ oh';, s)$;

$(p;, s) \rightarrow (psv.(ps.p), s)$;

$(\texttt{TICK};, s) \rightarrow (gc\ \texttt{:=}\ s.gc + 1;\ lc.p_1\ \texttt{:=}\ s.lc.p_1 + 1;\ \ldots;\ lc.p_n\ \texttt{:=}\ s.lc.p_n + 1;, s)$.

Activity predicates. They are defined by rules:

$(\texttt{(PROC}\ p\ \texttt{IN STATE ACTIVE)}, s) \rightarrow_R (s.ps.p \neq \texttt{STOP} \wedge s.ps.p \neq \texttt{ERROR}, s)$;

$(\texttt{PROC IN STATE ACTIVE}, s) \rightarrow_R (\texttt{(PROC}\ s.cp\ \texttt{IN STATE ACTIVE)}, s)$;

$(\texttt{(PROC}\ p\ \texttt{IN STATE INACTIVE)}, s) \rightarrow_R (s.ps.p = \texttt{STOP} \vee s.ps.p = \texttt{ERROR}, s)$;

$(\texttt{PROC IN STATE INACTIVE}, s) \rightarrow_R (\texttt{(PROC}\ s.cp\ \texttt{IN STATE INACTIVE)}, s)$;

$(\texttt{(PROC}\ p\ \texttt{IN STATE STOP)}, s) \rightarrow_R (s.ps.p = stop, s)$;

$(\texttt{PROC IN STATE STOP}, s) \rightarrow_R (\texttt{(PROC}\ s.cp\ \texttt{IN STATE STOP)}, s)$;

$(\texttt{(PROC}\ p\ \texttt{IN STATE ERROR)}, s) \rightarrow_R (s.ps.p = \texttt{ERROR}, s)$;

$(\texttt{PROC IN STATE ERROR}, s) \rightarrow_R (\texttt{(PROC}\ s.cp\ \texttt{IN STATE ERROR)}, s)$.

Control statements. They are defined by rules

$(\texttt{STOP PROC}\ p;, s) \rightarrow_R (lc.p\ \texttt{:=}\ 0;\ ps.p\ \texttt{:=}\ stop;, s)$;

$(\texttt{STOP};, s) \rightarrow_R (\texttt{STOP PROC}\ s.cp;, s)$;

$(\texttt{ERROR PROC}\ p;, s) \rightarrow_R (lc.p\ \texttt{:=}\ 0;\ ps.p\ \texttt{:=}\ error;, s)$;

$(\texttt{ERROR};, s) \rightarrow_R (\texttt{ERROR PROC}\ s.cp;, s)$;

$(\texttt{START PROC}\ p;, s) \rightarrow_R (lc.p\ \texttt{:=}\ 0;\ ps.p\ \texttt{:=}\ pso.p.1;, s)$;

$(\texttt{RESTART};, s) \rightarrow_R (\texttt{START PROC}\ s.cp;, s)$;

$(\texttt{SET STATE}\ \theta;, s) \rightarrow_R (lc.p\ \texttt{:=}\ 0;\ ps.p\ \texttt{:=}\ \theta;, s)$;

If $pso.p = con(\ldots, < s.ps.(s.cp), \theta >, \ldots)$ then $(\texttt{SET NEXT}, s) \rightarrow_R (lc.p\ \texttt{:=}\ 0;\ ps.p\ \texttt{:=}\ \theta;, s)$.

Timeout statements. They are defined by rules

$(\texttt{RESET TIMEOUT;}, s) \to_R (lc.(s.cp)\ \texttt{:= 0;}, s)$;

If $(e, s) \to_R (\gamma, s')$, and $s.cl.(s.cp) \geq \gamma$ then $(\texttt{TIMEOUT}\ e\ \eta, s) \to_R (\eta, s')$;

If $(e, s) \to_R (\gamma, s')$, and $s.cl.(s.cp) < \gamma$ then $(\texttt{TIMEOUT}\ e\ \eta, s) \to_R (\texttt{OK}, s')$.

General statements. They are defined by rules

If $(e, s) \to_R (\texttt{TRUE}, s')$ then $(\texttt{IF}\ e\ \eta_1\ \texttt{ELSE}\ \eta_2, s) \to_R (\eta_1, s')$;

If $(e, s) \to_R (\texttt{FALSE}, s')$ then $(\texttt{IF}\ e\ \eta_1\ \texttt{ELSE}\ \eta_2, s) \to_R (\eta_2, s')$;

If $(e, s) \to_R (\texttt{TRUE}, s')$ then $(\texttt{IF}\ e\ \eta, s) \to_R (\eta, s')$;

If $(e, s) \to_R (\texttt{FALSE}, s')$ then $(\texttt{IF}\ e\ \eta, s) \to_R (\texttt{OK}, s')$;

If $s.cf = f$, and $(e, s) \to_R (\gamma, s')$ then $(\texttt{RETURN}\ e\texttt{;}, s) \to_R (vv.val_f\ \texttt{:=}\ \gamma, s')$;

$(\texttt{\{}\eta_1\ \ldots\ \eta_m\texttt{\}}, s) \to_R (\eta_1\ \ldots \eta_m, s')$;

If $(\eta_1, s) \to_R (\texttt{OK}, s')$ then $(\eta_1\ \eta_2\ \ldots\ \eta_m, s) \to_R (\eta_2\ \ldots \eta_m, s')$.

Expressions. They are defined by rules

If $(e, s) \to_R (\texttt{FAIL}, s')$ then $(v\ \texttt{=}\ e, s) \to_R (\texttt{FAIL}, s')$;

If $(e, s) \to_R (\gamma, s')$ then $(v\ \texttt{=}\ e, s) \to_R (vv.v\ \texttt{:=}\ \gamma\texttt{;}, s')$;

If $par.f =< v_1, \ldots, v_m >$, $\texttt{FAIL} \notin \{\gamma_1, \ldots, \gamma_m\}$, and

$(e_1, s) \to_R (\gamma_1, s_1)$, $(e_2, s_1) \to_R (\gamma_2, s_2)$, ..., $(e_m, s_{m-1}) \to_R (\gamma_m, s_m)$

then $(f\texttt{(}e_1, \ldots, e_m\texttt{)}\texttt{;}, s) \to_R$

$(vv.v_1\ \texttt{:=}\ \gamma_1\texttt{;}\ \ \ldots\texttt{;}\ \ vv.v_m\ \texttt{:=}\ \gamma_m\texttt{;}\ bod.f, s_m)$;

If $par.f =< v_1, \ldots, v_m >$, $\texttt{FAIL} \notin \{\gamma_1, \ldots, \gamma_{k-1}\}$, $\gamma_k = \texttt{FAIL}$, and

$(e_1, s) \to_R (\gamma_1, s_1)$, $(e_2, s_1) \to_R (\gamma_2, s_2)$, ..., $(e_k, s_{k-1}) \to_R (\gamma_k, s_k)$

then $(f\texttt{(}e_1, \ldots, e_m\texttt{)}\texttt{;}, s) \to_R (\texttt{FAIL}, s_k)$.

Sequential composition. It is defined by rules

If $(\eta_1, s) \to_R (\texttt{OK}, s')$ then $(\eta_1\ \eta_2\ \ldots\ \eta_m, s) \to_R (\eta_2\ \ldots \eta_m, s')$;

If $(\eta_1, s) \to_R (\texttt{FAIL}, s')$ then $(\eta_1\ \eta_2\ \ldots\ \eta_m, s) \to_R (\texttt{FAIL}, s')$.

4. Discussion and Conclusion

Reflex has very simple semantics in terms of data structures, expressions and statements. It has no pointers, arrays, and structures. It has no iteration statements and jump statements. It has a limited number of operations. Its complexity is due to scan-based execution, interaction with environment, handling time intervals, process interaction and linking its variables to physical I/O signals. We propose operational semantics which copes with this complexity by using concepts of global and local clocks, dividing variables into internal, input and output ones, and modeling input/output of programs with history of values of input and output variables.

Current critical systems commonly use a lot of floating-point computations, and thus the

testing or static analysis of programs containing floating-point operators has become a priority. However, correctly defining the semantics of common implementations of floating-point is tricky, because semantics may change with many factors beyond source-code level, such as choices made by compilers [12]. Therefore, we plan to modify operational semantics rules for arithmetical operations based on the platform-independent approach to specification and verification of arithmetical operations and standard mathematical functions [13–15].

References

1. IEC 61131-3: Programmable controllers. Part 3: Programming languages. Rev. 2.0. International Electrotechnical Commission Std., 2003.
2. Basile F., Chiacchio P., Gerbasio D. On the Implementation of Industrial Automation Systems Based on PLC // IEEE Trans. on Automation Science and Engineering. 2013. Vol. 10, No. 4. P. 990–1003.
3. Travis J., Kring J. LabVIEW for Everyone: Graphical Programming Made Easy and Fun. 3rd Edition. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006
4. Zyubin V. Using Process-Oriented Programming in LabVIEW // Proc. of the Second IASTED Intern. Multi-Conference on automation, control, and information technology: Control, Diagnostics, and Automation. Novosibirsk, 2010. P. 35–41.
5. Randell B. Software Engineering Techniques // Report on a conference sponsored by the NATO Science Committee. Brussels, Scientific Affairs Division, NATO, Rome, Italy, 1970. P. 16.
6. Liakh T.V., Rozov A.S., Zyubin V.E. Reflex Language: a Practical Notation for Cyber-Physical Systems // System Informatics, No. 12. 2018. P. 85–104.
7. Norrish M. C formalised in HOL // Ph.D. thesis. University of Cambridge, Technical report, UCAM-CL-TR-453, 1998.
8. Gurevich Y., Huggins J. The semantics of the C programming language // Lecture Notes in Computer Science. 1993. Vol. 702. P. 274–308.
9. Blazy S., Leroy X. Mechanized semantics for the Clight subset of the C language // J. Autom. Reasoning. 2009. Vol. 43, No. 3. P. 263–288.
10. Nepomniaschy V.A., Anureev I.S., Mikhailov I.N., Promsky A.V. Towards verification of C programs. C-light language and its formal semantics // Programming and Computer Science. 2002. Vol. 28, No. 6. P. 314–323.
11. Ellison C., Rosu G. An Executable Formal Semantics of C with Applications // Proc. of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. 2012. P. 533–544
12. Monniaux D. The pitfalls of verifying floating-point computations // ACM Transactions on Programming Languages and Systems. 2008. Vol. 30, No. 3. P. 1–41.
13. Shilov N.V. On the need to specify and verify standard functions // The Bulletin of the Novosibirsk

Computing Center, Series: Computer Science. 2015. Vol. 38. P. 105–119.

14. Shilov N.V., Promsky A.V. On specification and verification of standard mathematical functions // Humanities and Science University Journal. 2016. Vol. 19. P. 57–68.
15. Shilov N.V., Anureev I.S, Kondratyev D., Promsky A.V. A summary of a case-study on platform-independent verification of the square root function in fix-point machine arithmetic // Proc. of 9th Workshop "Program semantics, specification and verification: theory and applications" (PSSV 2018). 2018. P. 85-91.