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 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 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 $(S, S_i, \rightarrow_R, \rightarrow_C)$, where S is a set of states, $S_i \subseteq S$ is a state of initial states, $\rightarrow_R$ and $\rightarrow_C$ 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 $\rightarrow_C$ 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, $lc \in P \rightarrow N$ is a function associating processes with their local clocks, all clocks count time in ticks (one tick corresponds to one iteration of control loop), $ps \in P \rightarrow \Theta$ is a function associating processes with their states, $vv \in V \rightarrow \Gamma$ is a function associating variables with their values, $ih \in V_i \rightarrow \Gamma^*$ 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), $oh \in V_o \rightarrow \Gamma^*$ 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: $\rightarrow_C \in \Xi_C \times \Xi_C$ and $\rightarrow_R \in \Xi_R \times \Xi$.

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

$$(u_1.u_2. \ ... \ .u_m \ \texttt{:=} \ u\texttt{;}, s) \rightarrow_R (u, upd(s, u_1.u_2. \ ... \ .u_m, u)).$$

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

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

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

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.

UDK 681.51, 004.05

Safety Analysis of Longitunal Motion Controllers during Climb Flight

Baar T. (Hochschule für Technik und Wirtschaft (HTW) Berlin, Department of Engineering I)

Schulte H. (Hochschule für Technik und Wirtschaft (HTW) Berlin, Department of Engineering I)

During the climb flight of big passenger planes, the pilot directly adjusts the pitch elevator and the plane reacts on this by changing its pitch angle. However, if the pitch angle becomes too large, the plane is in danger of an airflow disruption on the wings, which can cause the plane to crash. In order to prevent this, modern planes take advantage of control software to limit the pitch angle. However, if the software is poorly designed and if system designers have forgotten that sensors might yield wrong data, the software might cause the pitch angle to become negative, so that the plane loses height and can - eventually - crash.

In this paper, we investigate on a model for a Boeing passenger plane how the control software could look like. Based on our model described in Matlab/Simulink®, it is easy to see based on simulation that the plane loses height when the sensor for the pitch angle provides wrong data. For the opposite case of a correctly functioning sensor, our simulation does not indicate any problems. This simulation, however, is not a guarantee that the control is indeed safe. For this reason, we translated the Matlab/Simulink®-model of the controler into a hybrid program in order to make this system amenable to formal verification using the theorem prover KeYmaera.

Keywords: *Cyber-Physical System (CPS), Formal Safety Analysis, Hybrid Automaton*

1. Flight Control Model of Longitudinal Motion

For a complete description of the aircraft motion in the three dimensional space six variables are needed that denote the degrees of freedom of a rigid body. The aircraft motion is calculable by six nonlinear coupled ordinary differential equations (ODEs) of these variables. However, under certain assumptions, the ODEs can be decoupled and linearized into longitudinal and lateral equations. It is common practice to derive a third order state space model with the state vector

$$x = [\,\alpha \;\; q \;\; \theta\,]^T \tag{1}$$

to describe the longitudinal motion [4], [5]. The state vector contains (1) the angle of attack α, pitch rate q, and pitch angle θ (cmp. Fig. 1).

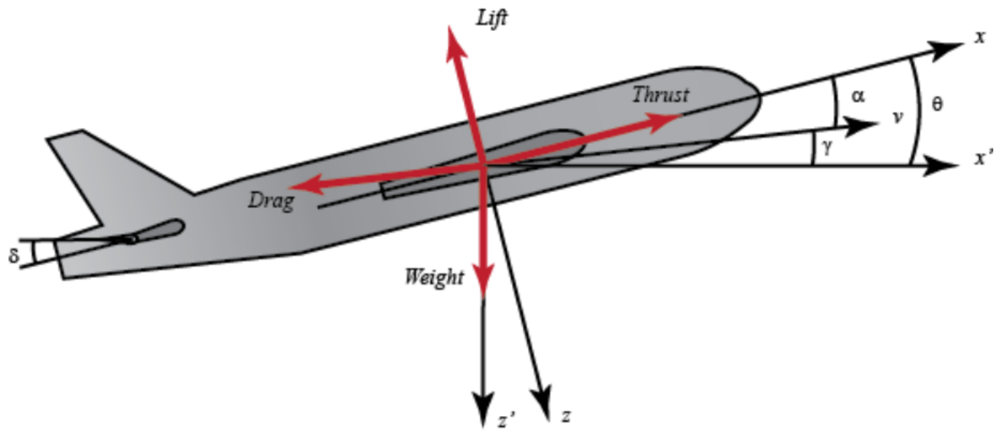


Fig. 1. Important parameters of Flight Model
(Source: *http://ctms.engin.umich.edu/CTMS/index.php?example=AircraftPitch &section=SystemModeling*)

Based on the assumption that the aircraft is in steady-cruise at constant altitude and velocity and that a change in the elevator deflection angle δ as controllable system input will not change the aircraft speed the longitudinal equations of motion for the aircraft in state space form $\dot{x} = f(x, u)$ with the state vector (1) and the input $u := \delta$ can be written as

$$\begin{aligned} \dot{\alpha} &= \mu\Omega\sigma\big[-(C_L + C_D)\alpha + \frac{1}{(\mu - C_L)}q - (C_W \sin\gamma)\theta + C_L\big] \\ \dot{q} &= \frac{\mu\Omega}{2I_{yy}}\Big(\big[C_M - \eta(C_L + C_D)\big]\alpha + \big[C_M + \sigma C_M(1 - \mu C_L)\big]q + (\eta C_W \sin\gamma)\delta\Big) \\ \dot{\theta} &= \Omega q \end{aligned} \tag{2}$$

where

$$\Omega = \frac{2U}{\bar{c}}, \qquad \mu = \frac{\rho S\bar{c}}{4m}, \qquad \sigma = \frac{1}{1 + \mu C_L}, \qquad \eta = \mu\sigma C_M, \tag{3}$$

with the equilibrium flight speed U and γ as the flight path angle. The parameter ρ denotes the density of air, S denotes the platform area of the wing, $\bar{c}$ denotes the average chord length and m denotes the mass of the aircraft, C_W denotes the coefficient of weight, C_M denotes coefficient of pitch moment, and I_{yy} as the normalized moment of inertia. The aerodynamic coefficients of thrust, drag and lift are C_T, C_D, C_L. Based on the above assumptions, the dynamics of the aircraft around a stationary operating point $p_c = (\alpha_c, q_c, \theta_c, \delta_c)$ for an equilibrium flight speed is obtained by Taylor linearization of $f(x, u)$

$$A = \frac{\partial f}{\partial x}|_{p_c} = \begin{pmatrix} -0.313 & 56.7 & 0 \\ -0.0139 & -0.426 & 0 \\ 0 & 56.7 & 0 \end{pmatrix}, \qquad B = \frac{\partial f}{\partial u}|_{p_c} = \begin{pmatrix} 0.232 \\ 0.0203 \\ 0 \end{pmatrix}$$

can be described as follows

$$\begin{aligned} \dot{\alpha} &= \quad -0.313\,\alpha + 56.7\,q + 0.232\,\delta \\ \dot{q} &= -0.0139\,\alpha - 0.426\,q + 0.0203\,\delta \\ \dot{\theta} &= \qquad\qquad\qquad\qquad 56.7\,q \end{aligned}$$

1.1. Control loop designed in Matlab/Simulink®

For the flight model introduced above, we have developed using Matlab/Simulink® a series of controllers those aim is to keep the pitch angle θ below a maximum value θ_{max} to prevent airflow disruption at the wings. We have selected the period of a climb flight and assume, that the pilot selects a constant δ_{man} as manual input, what might cause pitch angle θ to increase. If θ becomes greater than an upper bound, the anti-stall mode is activated and the controller computes a corrective δ_{corr} to prevent airflow disruption. When - as a consequence - θ falls again below a lower bound (due to hysteresis, the lower bound is slightly different from upper bound), the anti-stall mode is switched off again and the pilot's δ_{man} become again the input for the plane.

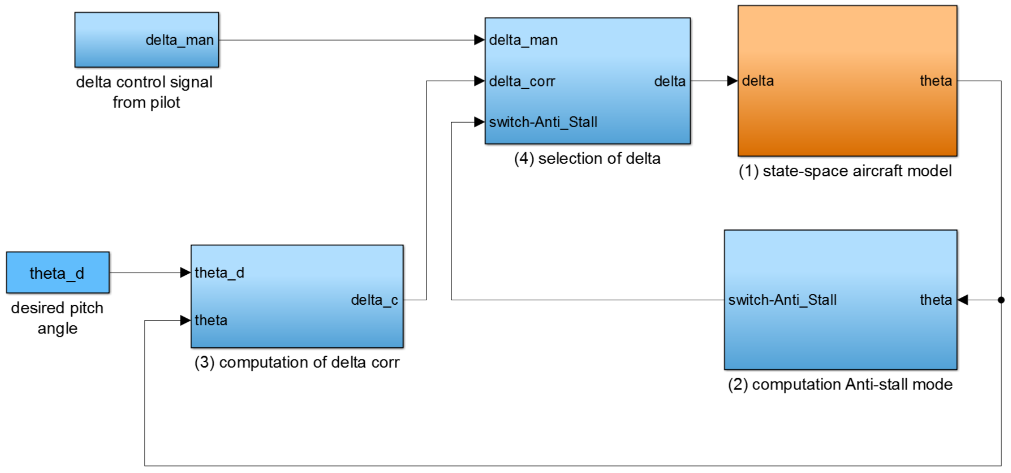


Fig. 2. Simple P-Controller designed in Matlab/Simulink®

A very simple version of the Matlab/Simulink®-controller is shown in Fig. 2 and consists of four main parts: (1) state space aircraft model, (2) computation of anti-stall mode, (3) computation of δ_{corr} (4) selection of δ from δ_{man}, δ_{corr} based on anti-stall mode.

1.2. System analysis in Matlab/Simulink®

The standard technique to analyze systems is by simulation, which is well-supported by Matlab/Simulink®.

1.2.1. Assuming correct sensor measuring for θ

The pitch angle θ is one of the outputs of the plant model and the input for the control loop. Based on θ, the input δ (pitch elevator angle) is computed for the next cycle.

Assuming that the angle θ is correcly measured by sensors, the simulation of the system does not show any situation in which θ becomes negative. This alone is not yet a guarantee, that this never happens but it is already a good starting point for formal verification of the system's safety (cmp. Sect. 2).

1.2.2. Assuming incorrect sensor measuring for θ

When building safety critical systems, engineers should always take into account that sensors might provided wrong data. We have modeled in a second Matlab/Simulink®-model a faulty sensor just by substituting the output θ of the plant model by $\theta + \theta_{offset}$. When simulating this second model, it can be immediately seen, that θ becomes soon negative, i.e. the plane can lose height.

2. Logical Analysis of Flight Control Models

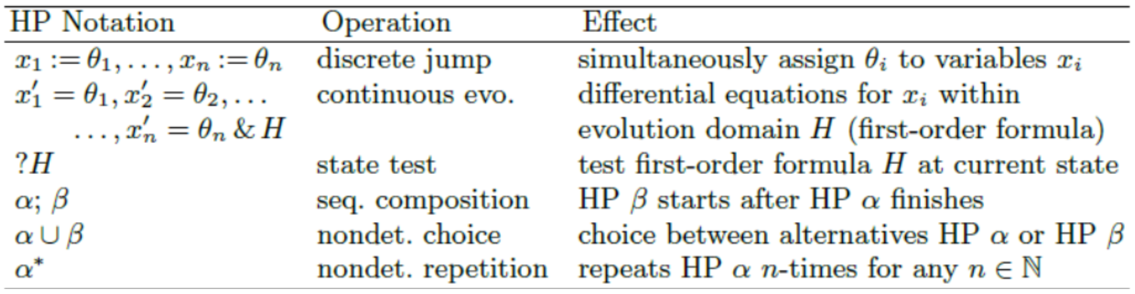
As detailed in the previous section, the Matlab/Simulink® toolkit is able to simulate the modeled system and it is easy to see, that the plane might lose height when sensors for measuring θ provide wrong data. However, for the opposite case of having a (presumably) correct system, simulation is not a sufficient technique in order to prove that the system behaves correctly under all possible circumstances. In our case, the correct behaviour means that θ remains always positive (recall that our system models the phase of a climb flight).

In this section, we present a translation of the Matlab/Simulink® model into a hybrid program (HP), a notion similar to well-known hybrid automata [2]. The notion of HP is supported by the theorem prover KeYmaera, which enables the user to formally verify safety properties of hybrid systems [3].

2.1. KeYmaera

A proof task for KeYmaera has to be formulated in differential dynamic logic (DDL), which is an extension of classical dynamic logic [1]. In short, classical dynamic logic is a modal logic with modalities *box* ($[\alpha]\psi$) and *diamond* ($<\alpha>\psi$). In the rest of the paper, only the box-modality is applied; the formula $[\alpha]\psi$ states that in each possible poststate after program α has terminated the formula ψ holds. Please note that termination of α is not claimed!

The main difference of DDL and DL is, that the former supports continuous state statements, in which variables changes its value automatically according to differential equations. This list of supported statements is summarized in the following table[1]. For a detailed introduction to DDL, the interested reader is referred to [3].



HP Notation	Operation	Effect
$x_1 := \theta_1, \ldots, x_n := \theta_n$	discrete jump	simultaneously assign θ_i to variables x_i
$x_1' = \theta_1, x_2' = \theta_2, \ldots$ $\ldots, x_n' = \theta_n \,\&\, H$	continuous evo.	differential equations for x_i within evolution domain H (first-order formula)
$?H$	state test	test first-order formula H at current state
$\alpha; \beta$	seq. composition	HP β starts after HP α finishes
$\alpha \cup \beta$	nondet. choice	choice between alternatives HP α or HP β
α^*	nondet. repetition	repeats HP α n-times for any $n \in \mathbb{N}$

2.2. Flight model as KeYmaera-Input

The Matlab/Simulink®-model shown in Fig. 2 can be translated into a hybrid program α as follows:

[1]The table has been taken from [3].

```
        {
            if      $theta > upper\_theta\_bound$  //adjust the mode
            then    $stallCtrl\_mode := 1$
            else    if $theta < lower\_theta\_bound$
                    then $stallCtrl\_mode := 0$
                    else skip
                    endif
            endif;
            $delta\_stall := (theta_d - theta) * k_p * stallCtrl\_mode;$
$\alpha =$  $delta := delta\_stall + delta\_manual * (1 - stallCtrl\_mode);$
            $t := 0;$
            {       $alpha' = -0.313 * alpha + 56.7 * q + 0.232 * delta,$
                    $q' = -0.0139 * alpha + -0.426 * q + 0.0203 * delta,$
                    $theta' = 56.7 * q,$
                    $t' = 1$
                    $\& t <= ep$
            }
        }*
```

Here, the control loop of the Matlab/Simulink®-model is encoded by an iteration (*) as the outermost operator of α. Within the iteration, we have basically a sequence ($cntrl; plant$), where *plant* is the continuous state statement $\{alpha' = \ldots \& t <= ep\}$ and *cntrl* is the sequence of statements before.

The statement *plant* directly corresponds to part (1) from Fig. 2. The statements forming *cntrl* realize the parts (2), (3), (4) from Fig. 2.

2.3. Proof task for correct behaviour

We can now formally formulate the safety property we would like to show for hybrid program α:

$$\theta > 0 \rightarrow [\alpha]\theta > 0 \tag{4}$$

In words, (4) reads as: whenever the system (including its controller) is started in a situation in which the pitch angle is positive, then after every control loop (which takes mostly time ep), the pitch angle remains positive. Note, that the proof of this claim will rely on some other assumptions, e.g. $ep > 0$, which have been suppressed here for the sake of brevity.

3. Conclusion and Future Work

In this paper, we have investigate safety critical software for controlling the flight of modern aircrafts. Such control software is usually developed using tools such as Matlab/Simulink®. We present a possible controller for the computation of the pitch elevator angle, but this controller has been completely designed by ourselves. Its sole purpose is to provide an example at which quality assurance techniques can be applied.

For the controller of our example, we review two main safety properties: Does the controller effectively prevent airflow disruption, which is the main purpose of the controller. However, there is another safety property, which can be easily overlooked when plane software is hastily developed: Is it possible that the controller software could cause the plane to lose height, which - eventually - might cause the plane to crash.

The simulations of our controller suggest, that both safety properties are met. However and not surprisingly, the controller can cause plane crashes when the sensor measuring the pitch angle θ does not provide correct data.

For the case that the sensor works correctly, we propose to translate the Matlab/Simulink®-model into a hybrid program and to apply the theorem prover KeYmaera in order to ensure for all possible situations that the system works correctly (not only for the few situations captured by the simulation).

Unfortunately, establishing such a formal proof using KeYmaera is a non-trivial task, because techniques for traditional software verification (e.g. case distinction for if-then-else) have to be combined with mathematical analysis methods for ordinary differential equations (ODEs). We will continue to work to formulate the key arguments of the proof so, that they can be processed by KeYmaera easily.

References

1. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Foundation of Computing. MIT Press, 2000.
2. Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings, 11th Annual IEEE*

Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996, pages 278–292. IEEE Computer Society, 1996.

3. Jan-David Quesel, Stefan Mitsch, Sarah Loos, Nikos Aréchiga, and André Platzer. How to model and prove hybrid systems with KeYmaera: A tutorial on safety. *STTT*, 18(1):67–91, 2016.
4. Robert F. Stengel. *Flight Dynamics.* Princeton University Press, 2004.
5. Thomas R. Yechout. *Introduction to Aircraft Flight Mechanics.* American Institute of Aeronautics & Astronautics, 2003.

UDK 004.822, 681.51

Constructing Verification-Oriented Domain-Specific Process Ontologies*

Natalia O. Garanina (A.P. Ershov Institute of Informatics Systems, Institute of Automation and Electrometry)

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

Vladimir E. Zyubin (Institute of Automation and Electrometry, Novosibirsk State University)

User-friendly formal specification and verification of concurrent systems from various subject domains are active research topics due to their practical significance. In this paper, we present the method for development of verification-oriented domain-specific process ontologies which are used to describe concurrent systems of subject domains. One of advantages of such ontologies is their formal semantics which makes possible formal verification of described systems. Our method is based on the verification-oriented process ontology. For constructing a domain-specific process ontology, our method uses techniques of semantic markup and pattern matching to associate domain-specific concepts with classes of the process ontology. We give detailed ontological specifications of these techniques. Our method is illustrated by the example of developing a domain-specific ontology for typical elements of automatic control systems.

Keywords: *process ontology, pattern matching, semantic markup, automatic control system, formal verification*

1. Introduction

Our long-term goal is a comprehensive approach to supporting formal verification of concurrent systems for ensuring their quality by formal methods. The solution includes methods for extracting formal models and properties of concurrent systems from the texts of technical documentation, as well as, instruments for manual correction of the extracted information and enriching it with new entities.

*The research has been supported by the Russian Ministry of Education and Science and the Russian Foundation for Basic Research (grants 17-07-01600, 19-07-00762).

Our envisaged intellectual system for supporting formal verification of concurrent systems will automatically extract and generate system requirements. We developed an Ontology of Specification Patterns as a first step towards creating this system [1]. Another key component of the system is the Process Ontology for concurrent systems [2]. The content of these ontologies, i.e. the sets of instances of their classes, are ontological descriptions of some concurrent system and requirements for it. These descriptions can be extracted from corpus of technical documentation by our system of information extraction from natural language text [3–5]. Such descriptions also can be developed by special editors which also can be used for correction of extracted information. These ontological descriptions for concurrent system processes and requirements are the basis for formal verification of the concurrent system because the Ontology of Specification Patterns and the Process Ontology have formal semantics. To verify a system, it is necessary first to choose a suitable verifier (model checker, in particular) taking into account the formal semantics of the ontology-based requirement presentation. If it exists, we translate the ontological description of the system into the model specification input language of the chosen verifier, and the requirements' description is translated into the property specification input language of this verifier (usually, this language is some temporal logic). Dealing with requirements in our system involves not only the formal semantics of specification patterns, but also the presentation of requirements both in a natural language and in a graphical form.

In this paper, we address both the problem of extracting a concurrent system description from technical documentation and developing editor for constructing and correcting the ontological description of concurrent systems. These tasks use the Process Ontology, which describes concurrent systems as consisting of communicating concurrent processes characterized by local and shared variables, and channels for communication by messages. This ontology has formal semantics based on labelled transition systems [2]. However, for requirement and verification engineers, the Process Ontology is very abstract to be suitable for supporting formal verification with our system.

Since this support system can be used for different subject domains, it is necessary to develop a method to specialize our abstract Process Ontology for specific subject domains in order to construct domain-specific processes instances which have variables and channels corresponding to their subject specialization. For example, in a concurrent system from the domain of Automatic Control System, the sensor-process must necessarily be connected by at least one communication channel with the process-controller. We must construct a Domain-

Specific Process Ontology to be a special case of the Process Ontology. Hence, this new ontology has formal semantics which makes possible formal verification of the systems it describes.

A Domain-Specific Process Ontology differs from the original Process Ontology in a set of axioms and rules that specify domain-specific restrictions on the attributes of the Process Ontology classes. This set of axioms has a declarative character. Ontology axioms can be used to check integrity and consistency of the ontology content. In case of the ontological concurrent system representation, integrity and consistency mean that instances of the ontology processes corresponding to processes of the subject domain have all necessary variables, channels and actions.

The declarative aspect of an ontology of domain-specific processes is suitable for checking the correctness of descriptions of already created or extracted concurrent systems. But for creating or correcting such a system, a constructive approach based on patterns of domain-specific processes is better. In this paper, we propose the method of constructing the domain-specific content of the Process Ontology using domain-specific patterns. The construction of this content includes several steps. First, we enrich classes of the Process Ontology (Section 2) with semantic markup attributes containing a string description of terms from a subject domain. The resulting new ontology called Semantically-Marked Process Ontology (Section 3) allows us to construct the domain-specific content of the Process Ontology. Then, patterns of domain-oriented processes are defined as instances of the Process-Oriented Semantic-Markup Patterns Ontology (Section 4). We illustrate our method with the example from the subject domain of Automatic Control Systems (Section 5). Development of the process ontology for this domain is especially important because a user-friendly formal specification and verification of automatic control systems, and, in general, cyber-physical systems have crucial practical significance.

2. Process Ontology

We consider an *ontology* as a structure, which includes the following elements: (1) a finite non-empty set of *classes*, (2) a finite non-empty set of *data attributes and relation attributes*, and (3) a finite non-empty set of *domains of data attributes*. Each class is defined by a set of attributes. Data attributes take values from domains, and relation attributes' values are instances of classes. *An instance of a class* is defined by a set of attribute values for this class. *A content* of an ontology is a set of instances of its classes.

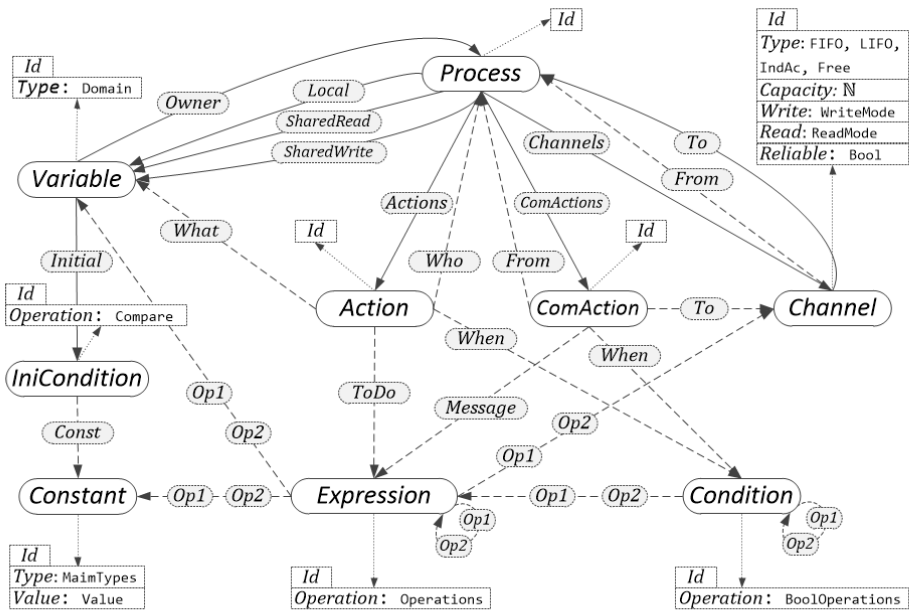


Fig. 1. Process Ontology.

The Process Ontology PO provides an ontological description of a concurrent system by a set of its instances. We consider a concurrent system as a set of communicating processes. Processes (described by the class Process) are characterized by sets of local and shared variables; a list of actions on these variables which change their values; a list of channels for the process communication; and a list of communication actions for sending messages. The process variables (the class $Variable$) and constants (the class $Constant$) take values in domains of basic types (Booleans, finite subsets of integers or strings for enumeration types) and finite derived types. Initial conditions of the variable values can be defined by comparison with constants. The actions of the processes (the class *Action*) include operations over variables' values. The enable condition for each action is a guard condition (the class $Condition$) for the variable values and the content of the channels. The processes can send messages through channels (the class $Channel$) under the guard conditions (the class $Condition$). The communication channels are characterized by the type of reading messages, capacity, modes of writing and reading, and reliability. Figure 1 represents the Process Ontology. Classes are presented by white ovals. Relations between classes are shown as dashed arrows with names in grey ovals. These arrows are solid if the relation is one-to-many, and dotted, if the relation is one-to-one. Class data attributes placed in dash-dot rectangles are connected with their classes by dash-dot arrows.

Classes of PO are universal because they do not take into account the features of a subject domain. In the next section, we define an extension of ontology PO — a semantically-marked process ontology that specifies necessary information about the subject domain.

3. Semantically-Marked Process Ontology

In this section, we formally describe our method of the semantic markup of the Process Ontology. This markup is used for matching the abstract processes of PO to specific processes of a chosen subject domain. The marking up is performed by enriching the classes of ontology PO with string labels corresponding to the concept of the subject domain. This classes with several service classes form the new semantically-marked process ontology. The instances of the subject domain processes can be constructed using this new ontology and the Process-Oriented Semantic-Markup Patterns Ontology described in the next section,

The *semantically-marked process ontology* ($SMPO$) contains domains $Classes$, $Domains$, $Types$, $Values$ corresponding to elements of PO ontology, domains $SLabel$, $SAttribute$, classes $AValue$, $Element$ and $Element\_T$ ($T \in Domains$) corresponding to semantic labelling, and classes of ontology PO enriched with semantic attributes based on listed new domains and classes.

Domains $Classes$ and $Domains$ include names of classes and domains from ontology PO. Domain $Types = Classes \cup Domains$ includes all names from ontology PO. Domain $Values$ includes all attribute values of PO: $Values = \cup_{T \in Types} Val(T)$, where $Val(T)$ is values of T, which are instances for $T \in Classes$ and the corresponding values for $T \in Domains$.

Domain $SLabel$ is a finite set of semantic labels which are strings. String labels specify information associated with the attribute values of ontology PO. This information can be about a subject domain (ex., "sensor" or "pressure") or special features of modeling processes (ex., "periodic start").

Domain $SAttribute$ is a finite set of semantic attributes which are string. Like labels, these semantic attributes specify subject domain information associated with the attribute values of ontology PO. The difference is that strings of the semantic attributes must be a string description of the attribute values of ontology PO (ex., "100", "$true$" or "instance of class $Controller$").

Further in class definitions, we add a superscript $*$ for multi-valued attributes and superscript 1 for mandatory single-valued attributes. Class $AValue$ (which instances are called attribute values) has two single-valued attributes: $Attribute^1$ with values in $SAttribute$ and $Value^1$ with values in $Values$ which specify the name of a semantic attribute and its value.

Class T of ontology $SMPO$ is some class of ontology PO enriched with two attributes: $SLabels^*$ with values in $SLabel$ and $SAattributes^*$ with values in $AValue$ (called *marking attributes*) which add the semantic markup to instances of class T. This attributes connect abstract notions of Process Ontology with a chosen subject domain. Attribute $SLabels$ specifies a set of semantic labels. Attribute $SAttributes$ specifies a set of semantic attributes with their values. Attribute $SAttributes$ cannot contain two instances of class $AValue$ with the same value of attribute $Atrribute$ for unambiguity of naming values in ontology PO. Attributes of T without the markup are called *base attributes*.

Class $Element\_T$ is constructed for each domain $T \in Domains$. This class has the marking attributes and attribute $Value^1$ with values in T. Attribute $Value$ specifies a value, and the marking attributes add the semantic markup to this value. Thus, in ontology $SMPO$, values of PO domains can be marked up.

Class $Element$ has only the marking attributes $SLabels^*$ and $SAattributes^*$. This class models new semantic classes (classes defined only by the semantic markup) in ontology $SMPO$. New semantic classes is used to construct new subject-oriented classes for ontology of processes in specific domains. This classes are used just for a readable description of a subject domain. They must be transformed to elements of ontology PO.

We illustrate addition of information about a subject domain to elements of ontology PO using the example of a sensor measuring temperature in degrees Celsius in the range from 0 to 1000. This sensor is specified by the following instance of class `Process` of $SMPO$ ontology:

```
Process(BAVs, SLabels:{"sensor"},
 SAttributes:{AValue (Attribute:"Dimension", Value:"temperature"),
  AValue(Attribute:"unit", Value:"Celsius"),
  AValue(Attribute:"range",
   Value:Element(SLabels:{"range"},
    SAttributes:{AValue(Attribute:"left", Value:"0"),
     AValue(Attribute:"right", Value:"1000")})})
```

Listing 1: Sensor instance

Here the tuple $T(A_1 : V_1, \ldots, A_n : V_n)$ denotes an instance of class T with values V_1, ..., V_n of attributes A_1, ..., A_n, the set $\{V_1, \ldots, V_n\}$ lists values of a multi-valued attribute, and $BAVs$ are base attributes from ontology PO.

Thus, with ontology $SMPO$ we can describe instances of notions from a subject domain by the semantic markup. However, this ontology is not enough to specify subject notions as elements of concurrent systems, i.e., to specify restrictions on sets of their instances. In the next section, we define a process-oriented ontology of semantic-markup patterns. This ontology is used to define notions of some subject domain using patterns by imposing restrictions on instances of classes of ontology $SMPO$ (what semantic markup can be added to them), as well as the arity and values of their attributes (the number of values of the attributes and what semantic markup can be added to these values).

4. Process-Oriented Semantic-Markup Patterns Ontology

Process-oriented semantic-markup patterns ontology ($POSMPO$) includes domains and classes of ontology $SMPO$, domains $AMatchSizes$ and $AMatchOperations$, and class $AMatch$.

Let n, m be nonnegative integers. Domain $AMatchArities = \{"m", "m|0", "mn", "mn|0", "m-", "m-|0", "-n"\}$ is used for restrictions on the number of attribute values of an ontology element matched with a pattern.

Domain $AMatchOperations = \{$"=", "<", "<=", "!=", ">", "=>", "in", "oneof", "all"$\}$ is a set of matching operations. They specify which values of ontology $SMPO$ must be matched to each other. The set of values of this domain can be extended for a specific subject domain.

Instances of $POSMPO$ classes are called *semantic-markup patterns*. Each pattern specifies a set of $SMPO$ instances matching with this pattern. Class T of ontology $POSMPO$ has the same attributes as class T of ontology $SMPO$, but they have values in $AMatch$. Class $AMatch$ specifies the rules for matching attribute values of $SMPO$ classes with patterns for them. This class has the following attributes: Ar with values in $AMatchArities$, Op with values in $AMatchOperations$ and Pat^* with values in $Values[SMPO]$ which contains all values of all attributes of all $SMPO$ classes similarly to domain $Values$ based on ontology PO. Attribute Ar restricts the number of matched values. Attribute Op defines the matching operation. Attribute Pat specifies patterns for attribute values.

Let $V.A$ denote the value of attribute A of instance V, and $|S|$ denotes the power of set S. We consider that instance V of class T from ontology $SMPO$ is matched with pattern P of class T from ontology $POSMPO$ iff for each attribute A of P such that $P.A = AMatch(Ar : R, Op : O, Pat : V_1, \ldots, V_n)$ the following holds:

1. If $R = "m"$ then $|V.A| = m$.
2. If $R = "m-"$ then $|V.A| \geq m$.
3. If $R = "-m"$ then $|V.A| \leq m$.
4. If $R = "m-k"$ then $k \leq |V.A| \leq m$.
5. If $R = "m|0"$ then $|V.A| = m$ or $|V.A| = 0$.
6. If $R = "m-|0"$ then $|V.A| \geq m$ or $|V.A| = 0$.

7. If $R = "m - n|0"$ then $n \leq |V.A| \leq m$ or $|V.A| = 0$.
8. If $O = " = "$ then $n = 1$ and $V' = V_1$ for each $V' \in V.A$. The cases when $O \in \{" = "," < "," <= ","! = "," > "," > "\}$ are defined similarly. This case restricts comparable attribute values.
9. If $O = "in"$ then $n = 1$ and $V' \in V_1$ for each $V' \in V.A$. This case defines membership of attribute values.
10. If $O = "oneof"$ then V is matched with $upd(P, A, V_i)$ for some $1 \leq i \leq n$, where $upd(Q, B, U)$ denotes the result of setting the value U to attribute B of instance Q. This case chooses some pattern value for the attributes.
11. If $O = "all"$ then there are S_1, ..., S_n such that $V.A = \{S_1, \ldots, S_n\}$, $S_i \cap S_j = \emptyset$ for $S_i \neq \emptyset$, $S_j \neq \emptyset$, and $upd(V, A, S_i)$ is matched with $upd(P, A, V_i)$ for each $1 \leq i \leq n$. This case chooses all pattern values for the attributes.
12. If O is undefined, and $A \neq SLabels$, or $T = AValue$ and $A \neq Attribute$, then V' is matched with $P.A$ for each $V' \in V.A$. This case reduces matching set of attribute values to matching separate attribute values of the set. The remaining cases are special ones for classes $SLabels$, $AValue$ и $SAttributes$.
13. If O is undefined and $A = SLabels$ then $P.SLabels \subseteq V.SLabels$.
14. If O is undefined, $T = AValue$, $A = Attribute$ then $n = 1$, and $V.A = V_1$.
15. If O is undefined and $A = SAttributes$ then $attributes(P.SAttributes) = attributes(V.SAtributes)$, where $attributes(AV)$ is the set of attributes in instance AV of class $AValue$.

We have defined a process-oriented ontology of semantic-markup patterns which combines the Process Ontology with descriptions of notions of a subject domain. A particular set of instances of this ontology gives the rules for constructing the corresponding subject-oriented process ontology. Classes and domains of $POSMPO$ provide a language for constructive using axioms which restrict abstract processes of PO with respect to a subject domain because these axioms can specify only numbers of attribute values and their ranges. In the next section, we construct some typical elements of Automatic Control Systems (ACSs) using classes $POSMPO$.

5. Domain-Specific Process Ontology for Typical Elements of Automatic Control Systems

In this section, we define semantic-markup patterns for typical elements of automatic control systems: simple and complex sensors, controllers, actuators and the controlled object.

Simple and complex sensors, and related entities are defined by patterns in Listing 2.

```
Process( // Simple sensor
Local:AMatch("0"),
SharedRead:AMatch("1", Variable(SLabels:{"Observed value"})),
SharedWrite:AMatch("0"),
Actions:AMatch("0"),
Channels:AMatch("1-",
Channel(SLabels:{"Channel from sensor to controller"})),
ComActs:AMatch("1-", ComAction(SLabels:{"Sending observed value from simple sensor"})),
SLabels:{"Simple sensor"},
SAttributes: {AValue("Physical quantity",
Element(SLabels:{"Physical quantity"})})

Element( // Physical quantity
SLabels:{"Physical quantity"},
SAttributes: {AValue("Dimension", AMatch(Op:"in", Pat:Dimension)),
AValue("Unit", AMatch(Op:"in", Pat:Unit)),
AValue("Range", Element{SLabels:{"Range"}})})

Element( // Range
SLabels:{"Range"},
SAttributes: {AValue("Left", AMatch(Op:"in", Pat:Float)),
AValue("Right", AMatch(Op:"in", Pat:Float))})

Variable( // Observed value
Users:AMatch(Op:"all",
Pat:{AMatch("1", Process(SLabels:{"Controlled Object"})),
AMatch("1-", Process(SLabels:{"Simple sensor"}))}),
SLabels:{"Observed value"},
SAttributes: {AValue("Physical quantity",
Element(SLabels:{"Physical quantity"})})

Channel( // Channel from sensor to controller
From:AMatch("1", "oneof", {Process(Slabels:{"Simple sensor"}),
Process(Slabels:{"Complex sensor"})}),
To:AMatch("1-", Process(SLabels:{"Controller"})),
Type:AMatch("1", "=", "FIFO"),
Capacity:AMatch("1", "=", 1),
Write:AMatch("1", "=", "Old"),
Read:AMatch("1", "=", "Keep"),
Reliable:AMatch("1", "=", "true"),
SLabels:{"Channel from sensor to controller"})

ComAction( // Sending observed value from simple sensor
From:AMatch("1", Process(Slabels:{"Simple sensor"})),
To:AMatch("1-", Channel(SLabels:{"Controller"})),
Message:AMatch("1",
Expression(Op1:AMatch("1",
Variable(SLabels:{"Observed value"})))
SLabels:{"Sending observed value from simple sensor"})

Process( // Complex sensor
SharedRead:AMatch("1-", Variable(SLabels:{"Observed value"})),
SharedWrite:AMatch("0"),
Channels:AMatch("1-",
Variable(SLabels:{"Channel from sensor to controller"})),
ComActs:AMatch("1-",
ComAction(SLabels:{"Sending message from complex sensor"})),
SLabels:{"Complex sensor"},
SAttributes: {AValue("Physical quantity",
Element(SLabels:{"Physical quantity"})})

ComAction( // Sending message from complex sensor
From:AMatch("1", Process(Slabels:{"Complex sensor"})),
To:AMatch("1-", Channel(SLabels:{"Controller"})),
SLabels:{"Sending message from complex sensor"})
```

Listing 2: Sensors

In this and the following listings, we use the following abbreviations: `SLabels:S` for `SLabels:AMatch(`

`Value:S)`, `AMatch(R, O, P)` for `AMatch(Ar:R, op:O, Pat:P)`, where `R`, `O`, or `P` can be omitted, `AMatch(R, O, P)` for `{AMatch(R, O, {P})}`, `AValue(A, V)` for `AValue(Attribute:A, Value:V)`, and `T` for `AMatch(Op:"in", Pat:T)`.

These patterns impose the following restrictions on sensors. Sensors must read observed values from variables shared with the controlled object and cannot change it. They must have outgoing channels connecting them with controllers and communication actions for sending messages to the controllers. There must be at least one controller and at least one shared variable associated with each sensor. Simple sensors have no local variables and actions whereas complex sensors have ones. Simple sensors can observe exactly one shared variable and send the observed value unchanged to controllers. For sensors, processing physical quantities must be defined. They are characterized by dimensions (`"temperature"`, `"pressure"`, `"density"`, etc.), units of measurement (`"centimeter"`, `"kilogram"`, `"volt"`, etc.) and ranges.

Controllers, *actuators* and *controlled objects* are restricted by patterns in Listing 3.

```
Process( // Controller
SharedRead:AMatch("0"), SharedWrite:AMatch("0"),
Channels:AMatch("all",
{AMatch("1-", Channel(SLabels:{"Channel from sensor to controller"}),
AMatch("0-", Channel(SLabels:{"Channel from actuator to controller"},
AMatch("1-", Channel(SLabels:{"Channel from controller to actuator"},
AMatch("0-", Channel(SLabels:{"Channel from controller to controller"})}),
ComActs:AMatch("1-", ComAction(SLabels:{"Sending message from controller"})),
SLabels:{"Controller"})

Process( // Actuator
SharedRead:AMatch("0"), SharedWrite:AMatch("0"),
Channels:AMatch("all",
{AMatch("1-", Channel(SLabels:{"Channel from controller to actuator"}),
AMatch("0-", Channel(SLabels:{"Channel from actuator to controller"},
AMatch("1", Channel(SLabels:{"Channel from actuator to controlled object"})}),
ComActs:AMatch("1-", ComAction(SLabels:{"Sending message from actuator"})),
SLabels:{"Actuator"})

Process( // Controlled object
SharedRead:AMatch("0"),
SharedWrite:AMatch("1", Variable(SLabels:{"Observed value"})),
Channels:AMatch("1-", Channel(SLabels:{"Channel from actuator to controlled object"}),
ComActs:AMatch("0"),
SLabels:{"Controlled object"})
```

Listing 3: Controllers, actuators and controlled objects

Controllers and actuators must not have shared variables. Controllers must have output channels connecting them with other controllers and actuators, and input channels connecting them with sensors and actuators. Actuators must have output channels connecting them with controllers and the controlled object, and input channels connecting them with controllers. There must be at least one sensor and at least one actuator connected with a controller through input and output channels, respectively. There must be at least one controller and the only controlled object connected with an actuator through input and output channels, respectively. The controlled object must be connected with actuators by input channels. There must be at least one shared variable, one sensor and one actuator associated with the controlled object.

Each pattern gives rules for defining an element of ACS in the Process Ontology. With a set of such patterns, we can specify a system of concurrent processes implementing typical elements of ACS. Thus, our method can be used to specify domain-specific processes.

6. Discussion and Conclusion

The method of developing a domain-specific process ontologies based on three core ontologies [6] has several remarkable properties. Verification-oriented process ontology PO specifies a compact universal process model with a labeled transition system as its formal semantics, which can be used in formal verification methods and model checking, in particular. Semantically-marked process ontology $SMPO$ makes possible marking instances of PO classes for associating them with concepts of a subject domain. Moreover, it is also possible to mark values of PO domains and describe new domain-specific classes. Process-oriented semantic-markup patterns ontology $POSMPO$ specifies restrictions on the semantic markup of instances of $SMPO$ classes, defining the subject concepts associated with these instances. Unlike the declarative approach describing a domain-specific process ontology by a set of axioms, this approach specifies the ontology as a set of patterns (instances of ontology $POSMPO$) for defining domain-specific processes constructively as instantiation of patterns from this set. All three ontologies are based on simple concepts that can be used as ontology design patterns [7, 8].

In the future, we plan to add new kinds of matching operations (for example, the current set of operations does not allow us to express the property that different attributes have the same instance as a value), to refine the process ontology for automatic control systems and to advance the method for building other domain-specific process ontologies.

References

1. Garanina, N., Zyubin, V., Lyakh, T., Gorlatch, S. An Ontology of Specification Patterns for Verification of Concurrent Systems // Proc. of the 17th Intern. Conf. SoMeT-18. New Trends in Intelligent Software Methodologies, Tools and Techniques. Series: Frontiers in Artificial Intelligence and Applications, Amsterdam: IOS Press, 2018. Vol. 303. P. 515-528. DOI 10.3233/978-1-61499-900-3-515.
2. Garanina N.O., Anureev I.S. Verification oriented process ontology // Proc. of 9th Workshop "Program semantics, specification and verification: theory and applications" (PSSV 2018). 2018. P. 58–67.

3. Garanina N.O., Sidorova E.A., Bodin E.V. A Multi-agent Text Analysis Based on Ontology of Subject Domain // Lecture Notes in Computer Science, 2015. Vol. 8974. P. 102–110.
4. Garanina N.O., Sidorova E.A. Context-dependent Lexical and Syntactic Disambiguation in Ontology Population // Proc. of the 25th Intern. Workshop on Concurrency, Specification and Programming (CS&P). Humboldt-Universitat zu Berlin, 2016. P. 101–112.
5. Garanina N.O., Sidorova E.A., Kononenko I.S., Gorlatch S. Using Multiple Semantic Measures For Coreference Resolution In Ontology Population // Intern. Journal of Computing. 2017. Vol. 16. No. 3. P. 166–176.
6. Scherp A., Saathoff C., Franz T., Staab S. Designing core ontologies // Applied Ontology, 2011. Vol. 6. No. 3. P. 177–221.
7. Gangemi A., Presutti V. Ontology Design Patterns // Staab, S., Studer, R. (eds.) Handbook on Ontologies. 2nd edn. Springer, 2009. P. 221–243.
8. Ontology design patterns. `http://www.ontologydesignpatterns.org`.

УДК 004.052.42

Towards automated error localization in C programs with loops[*]

Kondratyev D.A. (A.P. Ershov Institute of Informatics Systems SB RAS)

Promsky A.V. (A.P. Ershov Institute of Informatics Systems SB RAS)

The most recent trends in the C-light verification system are MetaVCG, semantic labels appropriate for verification condition (VC) explanation and symbolic method of definite iterations. MetaVCG takes a C-light program together with some Hoare's logic and produces on-the-fly a VC generator (VCG), which in turn processes the input program. Hoare's logic for definite iterations is a good choice if we try to get rid of loop invariants. Finally, if a theorem prover was unable to validate some VCs we could follow two ways. Obviously, we could revise/enrich specifications or/and underlying proof theory to prove the truth of VCs. Or, perhaps, we could concentrate upon establishment of falsity, which meant there were errors in annotated program. This is where semantic labels play crucial role providing some natural language comments about wrong VC as well as a back-trace to the error location. The newly developed ACL2 heuristics to prove VC falsity is the main theme of this paper.

Keywords: *deductive verification, semantic label, error localization, C-light, automated theorem proof, C-lightVer, ACL2, MetaVCG, symbolic method of verification of definite iterations, proof* strategy

1. Introduction

The C-light project [12] corresponds to the mainstream architecture of modern verification systems. It uses translation into an intermediate language (here, C-kernel) allowing to smooth over some hard corners of deductive verification. In order to improve efficiency we prefer domain specific verification condition (VC) generation, which means different generators for different program classes. Traditional approach implies manual reprogramming of VC generator (VCG). Instead, we adapted the MetaVCG approach of Moriconi and Schwarts [13]. For a given axiomatic system the MetaVCG automatically constructs an ordinary generator. The C-lightVer system is the implementation of the C-light project.

Are there any verification problems that cannot be solved by a two-level scheme or by

[*]This research is partially supported by RFBR grant 17-01-00789.

MetaVCG? Indeed, there are. The loop invariants, for example. Another approach adapted in our project, so called symbolic verification of definite iterations [14], proposes solution for certain class of cycles over data structures. The loop execution is modeled symbolically by replacement function `rep`. Some results of such adaptation were discussed in [6, 8].

At the moment, the main theorem prover for C-light is ACL2 [5]. The classic induction in ACL2 is not powerful enough to handle VCs with replacement function `rep`, so the proof strategies are proposed in [8, 11].

The traditional deductive verification works ideally for a priori correct Hoare triple with true VCs. If some VCs are false the user must analyze the prover signals to understand what went wrong and to localize possible errors. An easy task for toy examples which becomes a real problem for real programs. Denney and Fisher developed the semantic labeling approach [3]. For every VC it provides the proof protocol (in axiomatic semantics) as well as localizing information up to the level of separate terms. A natural language explanation can be generated from such protocol. The MetaVCG approach allowed us to easily introduce semantic labels in our axiomatic systems [6, 7, 12].

Now, after this background overview let us address the current problematic task, error localization for programs with loops. And again, situation is clear when ACL2 is able to discover truth or falsity. But often the answer is `"unknown"`. Instead of trying to satisfy a VC we can use some strategies to check unsatisfiability. Since all variables in ACL2 are implicitly universally quantified, the existence quantifier appears in the negated VC. Thus, our previous strategies [8, 11] fail here and we need revised ones. Another requirement — the `unsat` strategies must work for loops with abrupt termination (i.e. in presence of `break` statement). Possible solution of this problem is discussed in this paper.

Related work. Some proof strategies deserve mention. For example, the system `ACL2(ml)` [4] bases upon two methods. The first one is proof pattern recognition by means of statistical machine learning. The second one is symbolic searching for analogous lemma. However, the underlying theories may vary deeply, so machine learning is not very suitable for VC proof.

The SL-resolution is another well-known strategy [10]. Its inherent problem consists in necessary construction of useless resolvents. To oppose the growth of disjunct set the connection graph method was proposed [15]. In comparison to it, our strategy aims at the structure of literals, not disjuncts.

The goals of Constraint Logic Transformation project [2] remind ours. In the same time their strategies are suitable for predicate processing.

2. Methods used in C-light project

MetaVCG. The metagenerator takes proof rules and axioms as its input. Technically all of them represent patterns to be matched against C code [9]. The pattern language incorporates the first order logics and the C grammar. Expressions can contain nonterminal symbols, like uninterpreted predicate symbols or "fragment variables" [13] denoting code snippets. Affiliation of metadata with certain class in the pattern language is explicit [9]. For example, construction $any\_code(S)$ can be matched against any sequence (including empty) of C statements, construction $any\_predicate(P)$ can be matched against any predicate of specification language. Let vector $v = \langle v_1 \dots v_k \rangle$. Let each $expr_j (1 \leq j \leq k)$ is result of replacement of all occurrences of term $vector\_element$ in the expression $expr$ by v_j. Then construct $vector\_substitution(T,\ vec,\ expr)$ denotes the simultaneous replacement of all occurrences of each $v_j (1 \leq j \leq k)$ in the formula T by expression $expr_j$.

Semantic labeling. The idea of Denney and Fischer [3] consists in adding of semantic labels to the proof rules. Labels explain the result of rule application. We also use notation $\ulcorner t \urcorner^{l}$, which means that the term t is decorated with label l. Labels themselves take form $c(o, n)$, where c is a concept (the term role), o is a line range (in the source program) and n is an auxiliary information.

In contrast to the original idea, the labels in our VCs form hierarchy more suitable to explanation generation [6]. We perform the depth search in the label tree and for each label its common text is expanded by corresponding pattern filled by line numbers. The text patterns for every label concept are similar to the C format strings, they are also fed to MetaVCG.

In order to support arbitrary label concepts a special construction `(label t c)` was added to the language of proof rules [7]. Here, `t` is a term decorated by label and `c` is a string (label type).

Symbolic method for the definite iterations. Consider a program fragment of the form `for x in S do v := body(v, x) end`, where S is a data structure, x is a variable of type "element of S", v is tuple of the loop variables excluding x, *body* represents the loop body which does not alter x and terminates for every $x \in S$. The ways of modification of S are quite restricted. The loop body can only contain assignments, the `if` statements (perhaps nested) ant the `break` statements. We call such `for` loops definite iterations. Let v_0 be the tuple of values of variables from v at the loop entry point. Result of the whole definite iteration can be expressed by replacement operation $rep(v, S, body, n)$, where $rep(v, S, body, 0) = v_0$, $rep(i, v, S, body) = body(rep(i-1, v, S, body), s_i)$ for all $i = 1, 2, \ldots, n$. If the `break` statement was executed during the i-th iteration of the body, the whole definite iteration continues, though v does not change anymore: $\forall j (i \leq j \leq n)\ rep(i, v, S, body) = rep(j, v, S, body)$.

The MetaVCG allowed us to combine ideas of definite iterations and semantics labels in the form of the following pattern:

```
{P} prog {(vector_substitution(Q, v,
                        (label rep_iter rep(n, v, S, body).vector_element))}
|- {any_predicate(P)} any_code(prog)
for(int_var(i) = 0; int_var(i) < int_var(n); int_var(i)++)
admissible_construct(i, n, v, S, body)
{any_predicate(Q)}
```

where $admissible\_construct(i, n, v, S, body)$ corresponds to an admissible body of definite iteration, int_var corresponds to an integer variable. The construct $vector\_substitution(Q,\ v,\ rep(n,\ v,\ S,\ body).vector\_element)$ denotes the simultaneous replacement of all occurrences of each $v_t(1 \leq t \leq length(v))$ in the formula Q by $rep(n,\ v,\ S,\ body).v_t$. The recursive definition of admissible construct is described in [8]. The algorithms of matching these patterns and program constructs have been implemented in the C-lightVer system [6–9]. The inference rule for downward iteration is defined similarly.

Automated generation of replacement operation. The replacement operation generation is based on translation [8] of loop body constructs into ACL2. Consider, for example, construction $b*$:

$$(b*\ \ (\ldots(var\ expr)\ldots)\ result)$$

Expression $(var\ expr)$ denotes binding of variable var with the value of expression $expr$, which may depend on previously bound variables. The value of $b*$ is equal to value of $result$, which also can depend on bound variables. Values of variables from v correspond to values of members of structure fr of type $frame$. So, to model modification of some variable in v we bind object fr with new object which differs from the old one in the corresponding field. The abrupt termination is modeled by truth of boolean member *loop-break* of object fr. Instruction *break* is modeled by the following binding: $((when\ t)\ fr)$. Since condition *when* is true, such binding interrupts current block $b*$ and returns fr.

3. Proof strategy for formulas with replacement operation

The arguments of this strategy are implication ψ containing expression $rep(n, ...)$ and the premise ϕ.

In the beginning we try to prove formula

$$\phi \to rep(n, \ldots).loop\text{-}break \qquad (\psi\text{-}lemma\text{-}1)$$

by induction on n. If ACL2 succeeds, we add (ψ-*lemma-1*) to the underlying theory. This lemma means that premise ϕ of implication ψ represents situation when the loop is abruptly terminated. Then we try to prove ψ using (ψ-*lemma-1*) and induction on n.

If ACL2 failed to prove (ψ-*lemma-1*), we address to the formula

$$\phi \to \neg rep(n, \ldots).loop\text{-}break \qquad (\psi\text{-}lemma\text{-}2)$$

by induction on n. Again, if ACL2 validates (ψ-*lemma-2*) we add it to our theory in order to be used in the proof of ψ.

This strategy resembles one described in [8]. Indeed, both of them use the value of *loop-break* field. They are automatic. Finally, they are heuristics.

But differences also take place. First, this strategy is applied to any implication containing `rep`, whereas strategy from [8] analyses program postconditions only. Second, the latter one generates lemmas in the form of conjunction. The first conjunct is a VC and the second one establishes equality of antecedent from postcondition to the value of *loop-break*. Thus, lemmas generated in [8] have rather more complex structure.

4. UNSAT strategy for VCs

The argument ω of this strategy contains expression $rep(n, \ldots)$ and has the following form:

$$\forall x_1 \ldots x_n\ (\phi_1(x_1 \ldots x_n) \to \psi_1(x_1 \ldots x_n)) \wedge \ldots \wedge (\phi_m(x_1 \ldots x_n) \to \psi_m(x_1 \ldots x_n))$$

In ACL2 all variables of ω are implicitly universally quantified. Note that every ϕ_i or ψ_i is not forced to depend on all variables $x_1 \ldots x_n$. But for simplicity we imply such dependence.

Now let us prove $\neg\omega$:

$$(\exists x_1 \dots x_n \ (\phi_1(x_1 \dots x_n) \wedge \neg\psi_1(x_1 \dots x_n))) \vee \dots \vee \\ (\exists x_1 \dots x_n \ (\phi_m(x_1 \dots x_n) \wedge \neg\psi_m(x_1 \dots x_n)))$$

Let for each $i(1 \leq i \leq m)$ $T_i \equiv \exists x_1 \dots x_n \ (\phi_i(x_1 \dots x_n) \wedge \neg\psi_i(x_1 \dots x_n))$. To prove $\neg\omega$ it is sufficient to prove an arbitrary $T_i(1 \leq i \leq m)$. If we can find such T_i the answer of strategy is `VC is false`. Otherwise, the answer is `Unknown.`

Now, to the proof of T_i. Obviously truth of formula

$$T_i' \equiv (\exists x_1 \dots x_n \ \phi_i(x_1 \dots x_n)) \wedge (\forall x_1 \dots x_n(\phi_i(x_1 \dots x_n) \to \neg\psi_i(x_1 \dots x_n)))$$

is sufficient to prove T_i. Let us denote subformula $\exists x_1 \dots x_n \ \phi_i(x_1 \dots x_n)$ as U_i and the right subformula $\forall x_1 \dots x_n(\phi_i(x_1 \dots x_n) \to \neg\psi_i(x_1 \dots x_n))$ as V_i. We need to prove both U_i and V_i.

The process begins with interactive proof of U_i. Admittedly, sounds like undesirable step away from automated verification but the reasons are as follows. First, we use the weakest precondition calculus [8]. Therefore, U_i cannot contain replacement operation. Only the specification functions that are known to user take place in U_i. Second, almost all automatic proof assistants have problems with existence quantifier. Third, our heuristics rests upon hypothesis of simple antecedents and complex consequents in implications that were generated by the wp-calculus. Based on this assumption, we try to automatize a more complex proof of V_i. If user cannot validate U_i we suppose that the whole T_i is false.

If user confirms U_i, the proof of V_i starts. There are two possibilities. First, V_i may be free of function `rep`. We simply pass this universally quantified formula to ACL2. Second, V_i contains expression $rep(n, \dots)$. Anyway, since user does not know definition of `rep`, only the automated attempts are possible. The `break` statement can cause some trouble here because it complicates definition of `rep`. In this case, we address the strategy from Section 3.

5. Example

The goal of this experiment is to demonstrate execution of both strategies. This case study is also described in our on-line repository [1]. We deliberately made an error in the following function:

```
1. /*@ requires (0 < n) && (n <= len(a));
2.     ensures (grt-eql-cnt(n, key, a) == 0 ==> \result == 0) &&
3.            (grt-eql-cnt(n, key, a) > 0 ==> \result == 1)
4. */
5. int grt_eql_key(int n, int key, int a[]){
6.    int i, result = 0;
7.    for (i = 0 ; i < n; i++){
8.        if (a[i] < key){result = 1; break;}}
9.    return result;}
```

We suppose the reader is familiar with ACSL format of specifications. The logical function *grt-eql-cnt* counts the number of array elements greater or equal to `key`. Its definition is given in Appendix A. The program should look for array element that is greater than or equal to `key` and should return `1` or `0` according to the specification. Thus, the error is the use of operator `<` instead of operator `>=` in if-condition of loop body. The result of intermediate translation into C-kernel looks like:

```
 5. int grt_eql_key(int n, int key, int a[]){
 6.    /* begin changes Dec3 1 7-8 */
 7.    auto int i;
 8.    auto int result = 0;
 9.    /* end changes */
10.    for (i = 0 ; i < n; i++){
11.        if (a[i] < key){result = 1; break;}}
12.    return result;}
```

Note that neither specification nor definite iteration is modified by translator. The string `"begin changes Dec3 1 7-8"` stores data for the error localization protocol [12]. Of course, it did not appear because we knew about intentional error. It is inherent mechanism of our translation stage. In this particular case it declares that translation rule `Dec3` for declarations [12] was used and strings `7-8` are its result. Due to the symbolic method VCG produces single VC (vc-1):

$$\begin{gathered}\forall n, key, a\\ ((\ulcorner 0 < n \wedge n \leq len(a)\urcorner^{ass\_pre(1)} \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge\\ \ulcorner grt\text{-}eql\text{-}cnt(n, key, a) = 0\urcorner^{ass\_post(2)} \rightarrow\\ \ulcorner\ulcorner rep(n, key, a, 0).result\urcorner^{rep\_iter(10-11)} = 0\urcorner^{ens\_post(2)})\\ \wedge\\ (\ulcorner 0 < n \wedge n \leq len(a)\urcorner^{ass\_pre(1)} \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge\\ \ulcorner grt\text{-}eql\text{-}cnt(n, key, a) > 0\urcorner^{ass\_post(3)} \rightarrow\\ \ulcorner\ulcorner rep(n, key, a, 0).result\urcorner^{rep\_iter(10-11)} = 1\urcorner^{ens\_post(3)})\end{gathered}$$

where $IntArr$ is set of integral arrays. Function rep is defined in Appendix B. Semantic label ass_pre denotes hypothesis from precondition, ass_pre denotes hypothesis from postcondition, ens_post denotes goal from postcondition, rep_iter denotes substitution of replacement function [7]. As expected, vc-1 cannot be proved by SAT strategies, like those from [8, 11]. It is time to use our UNSAT strategy.

Formula vc-1 is a conjunction of two implications. Each of them uses function rep. The formula U_1 for the first conjunct ϕ looks like:

$$\begin{gathered}\exists n, key, a\\ 0 < n \wedge n \leq len(a) \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge\\ grt\text{-}eql\text{-}cnt(n, key, a) = 0\end{gathered}$$

Since the user knows definition of *grt-eql-cnt*, he can prove U_1 in interactive mode. Therefore, formula V_1 for conjunct ϕ appears:

$$\begin{gathered}\forall n, key, a\\ \ulcorner 0 < n \wedge n \leq len(a)\urcorner^{ass\_pre(1)} \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge\\ \ulcorner grt\text{-}eql\text{-}cnt(n, key, a) = 0\urcorner^{ass\_post(2)} \rightarrow\\ \ulcorner\ulcorner rep(n, key, a, 0).result\urcorner^{rep\_iter(10-11)} \neq 0\urcorner^{ens\_post(2)}\end{gathered}$$

This part of the proof is automatic. The presence of `break` in the loop body complicates things, so strategy from Section 3 enters on the scene. The corresponding V_1-*lemma*-1 is as follows:

$$\begin{gathered}\forall n, key, a\\ 0 < n \wedge n \leq len(a) \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge\\ grt\text{-}eql\text{-}cnt(n, key, a) = 0 \rightarrow\\ rep(n, key, a, 0).loop\text{-}break\end{gathered}$$

This lemma deserves further explanation. Since $0 < n$, the loop body is executed at least once. Amount of elements of a greater or equal to key is also zero. Thus, all elements in sub-array

$[0 : n-1]$ are less than *key*. Control expression of the *if* statement contains wrong operator, so for such array the *break* statement is executed. V_1-*lemma*-1 was proved automatically in ACL2 by induction on n and was added to the theory. Its `lisp` definition is given in Appendix C.

Finally, execution of `break` means that $result = 1$. So, V_1-*lemma*-1 contributes in automatic proof of V_1. The `lips` definition of V_1 is given in Appendix D. As a result *vc*-1 is false and explanation for V_1 is produced. Let us consider this explanation.

```
This formula corresponds to lines 1-9 in function "grt_eql_key".
Its purpose is to show unsatisfiable case. Hence, given
    - assumption that precondition from line 1 holds,
    - assumption that postcondition hypothesis from line 2 holds,
ensure that postcondition goal from line 2
    with substitution loop effect from lines 7-8 by rep
does not hold.
```

The error localization protocol [12] analyses semantic labels in V_1 and results of UNSAT strategy to generate the text above. Actually, labels in VCs contain string ranges of C-kernel program. This is where the commented information about translation rules proves to be useful. The location of possible error can be retranslated back to C-light program [12].

6. Conclusion

Many papers tend to demonstrate successful experiments avoiding the situation when verification fails. To fill this gap we discussed here some ideas about error localization in the C-light project. In case when the prover can confirm neither truth nor falsity, the traditional response if straightforward. User tries to reinforce the underlying theory in attempt to successfully reprove VCs. However, we do not discard bad scenario and simultaneously we try to check whether VCs are actually false. Another point of our current interest is verification of loops with `break` statement. We obtained some results in our experiments:

1. We devised the UNSAT strategy. It generates formula whose truth automatically implies falsity of original VC. At the moment such formula involves either interactive and automated proof.
2. We also propose the SAT strategy for VCs with operation `rep`. If abrupt termination of the loop is described in formula premises, a corresponding lemma will be added to the

underlying theory.

These strategies are the new contribution relative to previous research [6–8, 11]. Based on these methods, we conducted some experiments on error localization. We can also mention successful verification [6, 8] of function `asum` from the interface BLAS. The more complex data structures and the other functions from BLAS will be considered in future work.

References

1. Automated Error Localization in C Programs. URL: https://bitbucket.org/Kondratyev/verify-c-light. Last accessed 30 Apr 2019.
2. De Angelis E., Fioravanti F., Pettorossi A., Proietti M.: Verification of Imperative Programs by Constraint Logic Program Transformation // Electronic Proceedings in Theoretical Computer Science. Vol. 129. P. 186–210.
3. Denney E., Fischer B. Explaining Verification Conditions // Lecture Notes in Computer Science. B.: Springer-Verlag, 2008. Vol. 5140. P. 145–159.
4. Heras J., Komendantskaya E., Johansson M., Maclean E. Proof-Pattern Recognition and Lemma Discovery in ACL2 // Lecture Notes in Computer Science. B.: Springer-Verlag, 2013. Vol. 8312. P. 389–406.
5. Hunt W. A., Kaufmann M., Moore J. S., Slobodova A. Industrial hardware and software verification with ACL2 // Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences. 2017. Vol. 375. No. 2104. Article Number 20150399.
6. Kondratyev D. Implementing the Symbolic Method of Verification in the C-Light Project // Lecture Notes in Computer Science. Cham: Springer International Publishing AG, 2018. Vol. 10742. P. 227–240.
7. Kondratyev D.A. The extension of the C-light project using symbolic verification method of definite iterations // XVII All-Russian Conf. of Young Scientists on Mathematical Modeling and Information Technology. Computational technologies. 2017. Vol. 22. P. 44-59. (In Russian)
8. Kondratyev D. A., Maryasov I. V., Nepomniaschy V. A. The Automation of C Program Verification by Symbolic Method of Loop Invariants Elimination // Modeling and Analysis of Information Systems. 2018. Vol. 25. No. 5. P. 491–505. (In Russian)
9. Kondratyev D. A., Promsky. A. V. Developing a self-applicable verification system. Theory and practice // Automatic Control and Computer Sciences. 2015. Vol. 49. No. 7. P. 445–452.
10. Kowalski R., Kuehner D. Linear Resolution with Selection Function // Artificial Intelligence. 1971. Vol. 2. No. 3–4. P. 227–260.
11. Maryasov I. V., Nepomniaschy V. A., Kondratyev D. A. Invariant Elimination of Definite Iterations over Arrays in C Programs Verification // Modeling and Analysis of Information Systems. 2017. Vol. 24. No. 6. P. 743–754.
12. Maryasov I. V., Nepomniaschy V. A., Promsky A. V., Kondratyev D. A. Automatic C Program

Verification Based on Mixed Axiomatic Semantics // Automatic Control and Computer Sciences. 2014. Vol. 48. No. 7. P. 407–414.

13. Moriconi M., Schwarts R.L. Automatic Construction of Verification Condition Generators From Hoare Logics // Lecture Notes in Computer Science. B.: Springer-Verlag, 1981. Vol. 115. P. 363–377.
14. Nepomniaschy V.A. Symbolic method of verification of definite iterations over altered data structures // Programming and Computer Software. 2005. Vol. 31. No. 1. P. 1–9.
15. Siekmann J., Wrightson G. An Open Research Problem: Strong Completeness of R. Kowalski's Connection Graph Proof Procedure // Lecture Notes in Computer Science. B.: Springer-Verlag, 2002. Vol. 2408. P. 231–252.

A. The ACL2 definition of *grt-eql-cnt*

```
(define grt-eql-cnt ((n integerp) (key integerp) (a integer-listp))
    :guard-hints (("Goal" :induct (dec-induct n)))
    :returns (result natp :hints (("Goal" :induct (dec-induct n))))
    (b* (   (n (nfix n))
            (key (ifix key))
            (a (integer-list-fix a))
            ((when (zp n)) 0)
            ((when (< (len a) n)) 0))
        (if (<= key (nth (- n 1) a))
            (+ 1 (grt-eql-cnt (- n 1) key a))
            (grt-eql-cnt (- n 1) key a)))
    ///
    (fty::deffixequiv grt-eql-cnt))
```

Due to recursive nature of *grt-eql-cnt*, induction is preferable when it comes to the proof of statements containing this function.

B. The ACL2 definition of function *rep*

```
(fty::defprod frame ((loop-break booleanp) (i integerp) (result integerp)))

(fty::defprod envir ((lower-bound integerp) (key integerp) (a integer-listp)))

(define frame-init ((i integerp) (result integerp))
    :returns (fr frame-p)
    (make-frame :loop-break nil
        :i i
        :result result)
    ///
    (fty::deffixequiv frame-init))


(define envir-init ((lower-bound integerp) (key integerp) (a integer-listp))
    :returns (env envir-p)
    (make-envir :lower-bound lower-bound
        :key key
        :a a)
    ///
    (fty::deffixequiv envir-init))


(define rep ((iteration natp) (env envir-p) (fr frame-p))
    :measure (nfix iteration)
    :verify-guards nil
    :returns (upd-fr frame-p)
    (b* (   (iteration (nfix iteration))
            (env (envir-fix env))
            (fr (frame-fix fr))
            ((when (zp iteration)) fr)
            (fr (rep (- iteration 1) env fr))
            ((when (frame->loop-break fr)) fr)
            (fr (if (<  (nth
                            (- (+ iteration (envir->lower-bound env)) 1)
                            (envir->a env))
                        (envir->key env))
                    (b* (   (fr (change-frame fr :result 1))
                            (fr (change-frame fr :loop-break t))
                            ((when t) fr)) fr)
                    (b* ((fr fr)) fr)))
            ((when (frame->loop-break fr)) fr)
            (fr (change-frame fr
                     :i (+ iteration (envir->lower-bound env)))))
        fr))
```

C. The ACL2 definition of V_1-*lemma*-1

```
(defrule v-1-lemma-1
    (implies (and (< 0 n) (<= n (len a)) (integerp n) (integerp key) (integer-listp a)
            (= 0 (grt-eql-cnt n key a)))
        (frame->loop-break
            (rep n (envir-init 0 key a) (frame-init 0 0))))
    :enable (grt-eql-cnt envir-init frame-init rep)
    :hints (("Goal" :induct (dec-induct n))))
```

Construction of the form

```
    :hints (("Goal" :induct (dec-induct n)))
```

prompts ACL2 to use induction on n.

D. The ACL2 definition of lemma V_1

```
(defrule v-1
    (implies (and (< 0 n) (<= n (len a)) (integerp n) (integerp key) (integer-listp a)
            (= 0 (grt-eql-cnt n key a)))
        (not (= (frame->result
                    (rep n (envir-init 0 key a) (frame-init 0 0))) 0)))
    :enable (grt-eql-cnt envir-init frame-init rep v-1-lemma-1)
    :hints (("Goal" :induct (dec-induct n))))
```

UDK 004.052.42

Proving properties of Discrete-Valued Functions using Deductive Proof: Application to the Square Root

Todorov V. (Groupe PSA, France)

Taha S. (LRI, CentraleSupélec, Université Paris-Saclay, France)

Boulanger F. (LRI, CentraleSupélec, Université Paris-Saclay, France)

Hernandez A. (Groupe PSA, France)

For many years, automotive embedded systems have been validated only by testing. In the near future, Advanced Driver Assistance Systems (ADAS) will take a greater part in the car's software design and development. Furthermore, their increasing critical level may lead authorities to require a certification for those systems. We think that bringing formal proof in their development can help establishing safety properties and get an efficient certification process. Other industries (e.g. aerospace, railway, nuclear) that produce critical systems requiring certification also took the path of formal verification techniques. One of these techniques is *deductive proof.* It can give a higher level of confidence in proving critical safety properties and even avoid unit testing.

In this paper, we chose a production use case: a function calculating a square root by linear interpolation. We use deductive proof to prove its correctness and show the limitations we encountered with the off-the-shelf tools. We propose approaches to overcome some limitations of these tools and succeed with the proof. These approaches can be applied to similar problems, which are frequent in automotive embedded software.

Keywords: *Formal Methods, Deductive Proof, Proving Discrete-Valued Functions*

1. Introduction and Motivation

Today, the automotive industry relies mostly on a model-based approach for developing embedded software. It consists in connecting common library blocks (operators) to design and simulate a model of the behavior to be produced. It uses a higher level of abstraction than the code. Code with the behavior of the model is then produced automatically. The most commonly used tools for software design are Simulink, from the MathWorks, and Scade, from ANSYS.

The main advantage of this approach is that models can be simulated and debugged before code generation. Thus, some of the errors are found and fixed earlier in the design process. On the other hand, simulation shares many common points with testing and cannot prove that the calculation is correct. Furthermore, the implementation of a model on a specific hardware can bring behaviors that have not been seen before at design stage.

For the rest of the study we take as example a function calculating a square root. During the design stage, the simulation can use a standard implementation of this function. However, in the implementation, we replace it with an optimized version because of hardware constraints. Our example is a discrete-valued function implementing the square root calculation, which uses a linear interpolation table. In automotive applications, as on-board computers have limited power, discrete-valued functions are frequently used in the implementation to avoid complex calculations.

In the near future, we expect that authorities will require a certification for highly critical software in self-driving cars. Our motivation is to provide proofs of correctness for production code using formal methods.

In a previous paper [17], we summarized some experiments about applying tools that use formal methods to industrial software. In this paper, we give details about the application of deductive proof to production code, the problems we encountered with off-the-shelf tools, and some approaches to solve this type of problems. Our function has been implemented in C and we used Frama-C WP [12] for proving its correctness. As some of the goals were impossible to prove with Frama-C and its solvers we implemented it in SPARK (based on Ada) to prove it with GNATprove [5]. We discuss the results and how other methods such as Abstract interpretation can be combined with deductive proof.

2. Deductive methods

2.1. Preliminaries

The foundations of the proof of logical properties on an imperative language program were put forward by C. A. R. Hoare [11] in 1969. Based on the precise semantic of a computer program, Hoare proposed to prove certain properties by mathematical deductive reasoning, generally at the end of the program.

He introduced a notation called the *Hoare triple*, which associates a program Q, start hypotheses P, and expected output properties R:

$$P\ \{Q\}\ R$$

The logical meaning of this triple corresponds to: if P is true, then after executing program Q, R will be true if Q terminates. The calculus of Hoare's triples is, in general, undecidable.

The proving by application of Hoare's rules is an intellectual process and is not tool driven. It is up to the author of the proof to define the correct properties between each instruction of the program and to establish its demonstration by applying the different theorems. This activity is not adapted to process thousands of lines of code in an acceptable time.

An initial automation of the process of proving programs was brought by the calculation of the WP (*Weakest Precondition*) from Dijkstra [8]. The principle consists in automatically calculating the most general property *WP(S,P)* holding before a statement S such that property P holds after the execution of S:

$$WP(S,P)\ \{S\}\ P$$

The calculus of WP is defined for each instruction. The proof process consists in calculating WP by going backward from the end of the program for which we want to prove P, up to the beginning. For full correctness, S must terminate.

The returned predicate from the WP calculation can rapidly become rather complex. Efficient (quadratic instead of exponential) verification condition generation (including WP generation) were proposed in the following papers [2, 10, 16]. In order to automate the process, all modern tools based on WP are using automatic theorem provers as back-end. We can cite, for example Alt-Ergo [6], Colibri[1], CVC4 [3], Yices2 [9], Z3 [7].

2.2. Tools for deductive reasoning

As we are interested in tools used by the industry, we present here only those that are mostly used today: Atelier B[2], Caveat [15], Frama-C WP and GNATprove.

2.2.1. Atelier B

Atelier B is a tool supporting the B method, which is a formal methodology to specify, build and implement software systems. The B method was originally developed in the 1980s by Jean-Raymond Abrial [1] and is based on first-order logic, set theory, abstract machine theory and refinement theory. This method is suitable for a new development. As we reused existing C code, we did not use this method.

[1]Colibri: `http://smtcomp.sourceforge.net/2018/systemDescriptions/COLIBRI.pdf`

[2]Atelier B: `https://www.atelierb.eu`

2.2.2. Caveat and Frama-C WP

Caveat and Frama-C WP are tools for deductive reasoning on C programs. Caveat was introduced at Airbus in 2002 to replace unit tests by unit proof and thus obtain a cost reduction and quality improvement over this part of the development process. The tool with its back-end Alt-Ergo were certified and recognized by the aviation certification authorities. Caveat analyzed C programs (with some restrictions in terms of language constructs) and had its own specification language based on a first order logic.

Frama-C is the academic open source tool developed by the same team as Caveat. Its WP module verifies properties written in the ACSL[3] language in a deductive manner. It implements the Weakest Precondition calculus and targets multiple automatic solvers via the Why3 platform[4].

2.2.3. GNATprove

GNATprove is a tool for deductive reasoning over SPARK (based on Ada) programs. Like Frama-C, it uses the Why3 platform but SPARK supports bit-vector data types. A bit-vector is an array data type for compactly storing bits. Most modern SMT-solvers support a theory of bit-vectors, which can help solving problems using this data type. Furthermore, for properties that are not valid, GNATprove can obtain a counterexample from the SMT solver.

3. Experiment

We took the C code implemented in an on-board computer to prove its correctness using deductive proof. The function calculates the square root Y of X by linear integer interpolation between two known points (X_a, Y_a) and (X_b, Y_b) using the following formula:

$$Y = Y_a + (X - X_a)\frac{(Y_b - Y_a)}{(X_b - X_a)}$$

This code is used in an implementation on an on-board computer, which cannot use floating-point numbers. We calculate the square root for numbers between 0.00 and 100.00 using an integer representation. We consider it as a fix-point number (multiplied by 100 to have a precision of 2 digits after the decimal separator), thus the input range is between 0 and 10000 (representing 0 and 100.00) and the returned result is a linearly interpolated value between 0 and 1000 (to be interpreted as a number between 0 and 10.00). We want to prove that the

[3]ACSL specification language: `https://frama-c.com/acsl.html`

[4]Why3: `http://why3.lri.fr/`

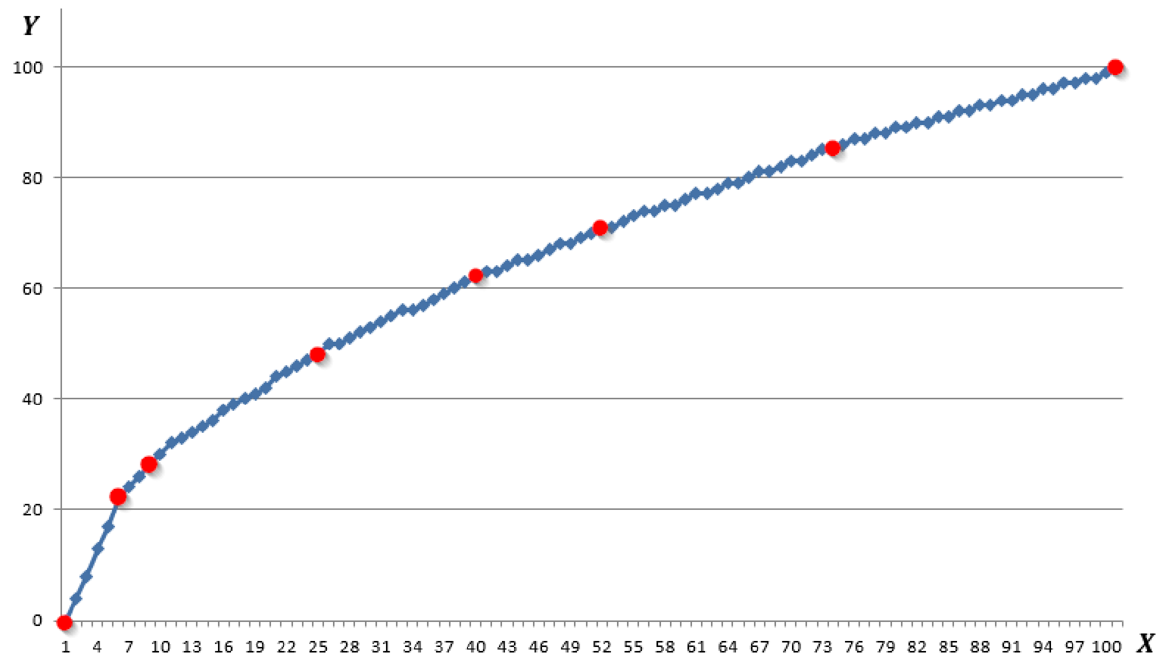


Fig. 1. Square root calculation in [0, 1.00] by linear interpolation from eight values

calculation is correct for a given precision. We present an example for the calculation in the [0, 1.00] subrange using eight known values in Fig. 1.

We proceeded in two steps. First, we proved a simplified version of the code using only eight values in the interpolation table (Fig. 1). These values were a subset of the full table present in the code, which contains 41 values. Then, we added the other values in the table and updated the contracts to take into account the new bounds. At our surprise, this did not scale up with Frama-C. We worked with the developers of Frama-C to understand why (we explain it in Section 4). Then we rewrote the function in SPARK[5] to see whether it would scale better. The main difference between C and SPARK is that we can specify a `bit-vector` data type in SPARK. For our use case, it helped the solver to reason using modular arithmetic. Most SMT solvers used as back-end via Why3 have a theory of bit-vectors. If we do not use bit-vectors, the SMT solver is reasoning by default using non-modular arithmetic. We also analyzed our complete C code with Astrée [13] from AbsInt, a static analysis tool using abstract interpretation, to prove some difficult goals and provide useful hypotheses to Frama-C WP.

Our first proof on the simplified code succeeded with Frama-C. Extending the table to 41 values, as in the real code, did not succeed. On the other hand, SPARK succeeded with the full table of 41 values.

[5]Special thanks to Yannick Moy from AdaCore

4. Results

In this section, we explain the results and why Frama-C failed to scale-up from 8 to 41 values, and what should be done to cope with this type of problems.

4.0.1. From Frama-C to the SMT solver

To understand the reason why automatic proof failed for the full table, we have to detail the transformations between the C code through Frama-C, Why3 and the solvers. First, Frama-C transforms the C code and its ACSL contracts using the weakest precondition calculus into verification conditions (VC) in the WhyML language. It also introduces additional goals to verify the absence of runtime errors such as overflows. The WhyML output contains all the theories necessary for the proof and is sent to Why3. Then Why3 transforms it into the language of the chosen prover. For our use case, the WhyML transformation contained quantified formulas and redefined some operators such as `division` using uninterpreted functions.

4.0.2. The difficult goal

There were 51 goals (verification conditions) to be proved and two of them were not proven. The most difficult goal was about proving that the contract of the post condition in the linear interpolation function had the same behavior as the code. We show it in Fig. 2.

Actually, contracts use mathematical arithmetic (without overflow), but code uses modular arithmetic, where overflows may occur. For our use case, we used a 16-bit unsigned integer to store the returned value of the interpolation.

4.0.3. Direct proof with SMT-LIB

Since 2 goals were not proven with the official Frama-C version, we obtained a new version that could address directly SMT solvers using the SMT-LIB standard [4]. We proved our goals with Colibri, CVC4 and Yices2. We remarked that the SMT-LIB file did not contain quantifiers and did not redefine operators such as `division`. We concluded that this approach scaled and worked better for problems with nonlinear arithmetic such as interpolation functions. Furthermore, some SMT solvers such as Yices2 do not support quantification.

4.0.4. Experience with the Why3 SMT output files

We wanted to understand what was the impact of the redefined division using uninterpreted functions and of quantified formulas, so we modified manually the SMT request sent to the solver. First, we removed the specific functions about division and used the standard SMT-LIB `div` operator. Then, the proof succeeded with CVC4 but only if using nonlinear logic containing bit-vectors. Disabling bit-vectors from that logic resulted in a failure to prove the formula. On the other hand, the quantifier-free SMT output did not need bit-vector logic to be proved.

4.0.5. Abstract interpretation

Because it is difficult to understand how the SMT solvers proved the difficult goal, we used Astrée to prove the absence of overflow in the returned value of the linear interpolation function. This proof can then be used as hypothesis in Frama-C WP. Astrée could find the dependency between Y_b and Y_a and estimate a precise interval for $(Y_b - Y_a)$. The same was done for $(X_b - X_a)$ and $(X - X_a)$. Thus a precise interval was calculated for Y in [0, 10000], which fits in a 16-bit unsigned integer without overflow.

5. Methodology

In this section, we propose a methodology based on our experience to solve problems using discrete-valued functions such as linear interpolation. Our use case is a simple one and we could have tested it for each value in the domain of validity of the function. However, in practice, there are more complex discrete-valued functions implemented with linear interpolation tables

```
typedef unsigned short uint16;
typedef unsigned char uint8;
/*@
  requires 0 <= Xa <= 10000 && 0 <= Xb <= 10000;
  requires 0 <= Ya <= 1000 && 0 <= Yb <= 1000;
  requires Yb > Ya && Xb >= Xa;
  requires Xa <= X <= Xb;
  ensures Xa != Xb ==> \result == (Ya + (X - Xa) * (Yb - Ya) / (Xb - Xa));
  ensures Xa == Xb ==> \result == Ya;
  assigns \nothing;
*/
uint16 LinearInterpolation(uint16 Xa, uint16 Ya, uint16 Xb, uint16 Yb, uint16 X)
{
  if (Xa != Xb) {
    return(Ya + (X - Xa) * (Yb - Ya) / (Xb - Xa));
  } else {
    return(Ya);
  }
}
```

Fig. 2. Annotated interpolation function for Frama-C WP automatic proof

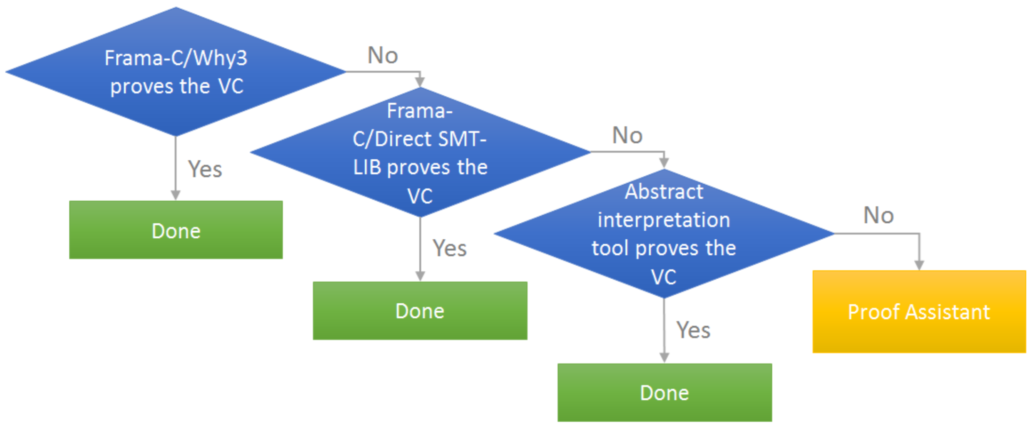

Frama-C/Why3 proves the VC
No
Yes
Done
Frama-C/Direct SMT-LIB proves the VC
No
Yes
Done
Abstract interpretation tool proves the VC
No
Yes
Done
Proof Assistant

Fig. 3. Methodology for proving Discrete-Valued Functions

called lookup tables. These functions are often called by other discrete-valued functions. The number of cases to test can be the product of the cardinalities of the domains of the individual functions. We propose to use the methodology shown below in Fig. 3 in order to prove those functions.

First, we need to isolate all the functions we want to prove together and annotate the code with contracts specifying the behavior expected from each function. Then, we can try to prove it in Frama-C via Why3. If the proof succeeds, we can stop. Otherwise, we can try to use the direct SMT-LIB output of Frama-C WP with the SMT solvers. As we have seen, this approach removes quantifiers and uses native mathematical operators. If it does not succeed, for some goals (VCs) we can try to prove them using abstract interpretation tools. If this method does not succeed, we need to use a proof assistant to prove the difficult goals.

6. Conclusions

In this paper, we have presented our experiments with automatic deductive proof of correctness of a discrete-valued function calculating a square root by interpolation. We used Frama-C WP and GNATprove to prove the function correct but encountered some difficulties with the nonlinear formula of the linear interpolation. Three non-standard approaches worked well for us: the use of bit-vectors in SPARK, the direct SMT-LIB quantifier-free output of Frama-C and the static analysis with Astrée. Bit-vectors are well supported in most modern SMT solvers and are well suited for problems that involve modular arithmetic, but scaling is sometimes difficult. For our use case, SMT requests without quantifiers performed and scaled better because there was no need for bit-vectors. Abstract Interpretation analysis gave more confidence in proving that there was no overflow in the linear interpolation calculus. We have proposed a methodology to use a combination of these different methods until the proof is done. We also show that using industrial use cases with off-the-shelf tools does not always scale, but if we work with researchers, we can find a solution and improve the tools.

Using deductive methods is very promising in an industrial context for safety-critical applications. It can replace unit tests as shown in [14] and thus decrease cost while increasing quality. It is also an intellectual activity that brings more satisfaction for engineers compared to testing.

References

1. Abrial J.R. The B-book: assigning programs to meanings. 1996.
2. Barnett M., Leino K.R.M. Weakest-precondition of Unstructured Programs // SIGSOFT Softw. Eng. Notes. 2005. Vol. 31. No. 1, P. 82–87.
3. Barrett C., Conway C.L., Deters M., Hadarean L., Jovanovi'c D., King T., Reynolds A., Tinelli C. CVC4 // In: Gopalakrishnan G., Qadeer S. (eds.) (CAV'11). LNCS. Springer, 2011. Vol. 6806, P. 171–177.
4. Barrett C., Stump A., Tinelli C., Boehme S., Cok D., Deharbe D., Dutertre B., Fontaine P., Ganesh V., Griggio A., Grundy J., Jackson P., Oliveras A., Krstić S., Moskal M., Moura L.D., Sebastiani R., Cok T.D., Hoenicke J. The SMT-LIB Standard. Version 2.0. Tech. rep. 2010.
5. Chapman R. Industrial Experience with SPARK // Ada Letters. 2000. Vol. XX. No. 4.
6. Conchon S., Coquereau A., Iguernlala M., Mebsout A. Alt-Ergo 2.2. // SMT Workshop: International Workshop on SMT. Oxford, United Kingdom, 2018.
7. De Moura L., Bjorner N. Z3: An Efficient SMT Solver // TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008.
8. Dijkstra E.W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs // ACM. 1975. Vol. 18. No. 8. P. 453–457.
9. Dutertre B. Yices 2.2 // In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. Springer, Cham, 2014. P. 737–744.
10. Flanagan C., Flanagan C., Saxe J.B. Avoiding Exponential Explosion: Generating Compact Verification Conditions // SIGPLAN Not. 2001. Vol. 36. No. 3. P. 193–205.
11. Hoare C.A.R. An Axiomatic Basis for Computer Programming. 1969.
12. Kirchner F., Kosmatov N., Prevosto V., Signoles J., Yakobowski B. Frama-C: A software analysis perspective // Formal Aspects of Computing. 2015. Vol. 27. No. 3.
13. Mauborgne L. Astrée: Verification of Absence of Runtime Error // In: Jacquart R. (ed.) Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27, Toulouse, France, Springer, 2004. P. 385–392.
14. Moy Y., Ledinot E., Delseny H., Wiels V., Monate B. Testing or Formal Verification: DO-178c Alternatives and Industrial Experience // IEEE Soft. 2013. Vol. 30. No. 3.

15. Randimbivololona F., Souyris J., Baudin P., Pacalet A., Raguideau J., Schoen D. Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach // Proc. of the Wold Congress on Formal Methods in the Development of Computing Systems. Springer-Verlag, London, 1999.
16. Shilov N.V., Anureev I.S., Bodin E.V.: Generation of correctness conditions for imperative programs // Programming and Computer Software. 2008. Vol. 34. No. 6. P. 307–321.
17. Todorov V., Boulanger F., Taha S. Formal Verification of Automotive Embedded Software // Proc. of the 6th Conf. on Formal Methods in Software Engineering. FormaliSE'18, ACM, New York, USA, 2018. P. 84–87.

UDC 519.7

On parallel composition of Finite State Machines with timed guards

Tvardovskii A.S. (Tomsk State University),

Yevtushenko N.V. (Ivannikov Institute for System Programming of the RAS)

Finite State Machines (FSMs) are widely used for analysis and synthesis of digital components of control systems. In order to take into account time aspects, timed FSMs are considered. In this paper, we address the problem of deriving a parallel composition of FSMs with timed guards and output delays (output timeouts). When the parallel composition is considered, component FSMs work in the dialog mode and the composition produces an external output when interaction between components is terminated. In this work, we formally introduce the parallel composition operator for FSMs with timed guards (TFSM) and show that unlike classical FSMs, a "slow environment" and the absence of live-locks are not enough for describing the behavior of a composition of deterministic TFSMs by a deterministic FSM with a single clock. Although the set of deterministic FSMs with timed guards is not closed under the parallel composition operator, some classes of deterministic TFSMs are still closed under this operator.

Keywords: *Timed Finite State Machines, output timeouts, timed guards, parallel composition*

1. Introduction

Finite State Machines (FSMs) are widely used for analysis and synthesis of discrete systems [1, 2], for example, for test derivation [3], verification [4] and optimization [1]. An FSM models a discrete event system that moves from state to state producing an output when an input is applied and is appropriate when reactive systems are considered. When the behavior of a complex system is described, a hierarchical approach is usually applied, i.e., a complex system is represented as a system of interacting components which usually are ‘simpler’ than the whole system. In the context of the FSM theory, a number of composition operators are considered and the notions of the parallel and synchronous compositions are defined [2]. A parallel composition describes the behavior of component machines which interact in a dialogue mode and is known to be appropriate when describing the joint functioning of protocol implementations and/or software components of telecommunication systems under the assumption of ‘one message in transit’. The parallel composition operator is well known for classical FSMs and the set of deterministic FSMs is closed under this operator when a so-called slow environment is considered. The slow environment means that the next input can be applied to the FSM composition only after the composition has produced an external output to the previous input.

Nowadays, a number of reactive systems take into account time aspects and thus, FSM based methods have to be extended to FSMs with time aspects such as timed guards, input and output timeouts [5, 6, 7]. A timed FSM can be considered as a good compromise between the expressive power and simplicity when considering timed automata [4, 8]. An FSM with timeouts can spontaneously change the current state if no input is applied before the input timeout expires. An FSM with time guards can have time restrictions at a state and thus, the behavior depends on a time instance when an external input is applied at the current state. Output timeouts in both cases describe the number of time units needed for processing an applied input.

In order to describe the joint behavior of communicating machines with time aspects, the composition operator has to be defined for the collection of Timed FSMs (TFSM). The parallel composition operator for FSMs with timeouts was proposed in [9] and is based on the intersection of corresponding underlying automata derived for TFSM components.

In this paper, we propose an algorithm for deriving a composed machine for a system of communicating FSMs with timed guards, namely, we define the parallel composition operator for a pair of such complete deterministic FSMs. It is shown that unlike classical FSMs, the set of deterministic FSMs with timed guards is not closed under the parallel composition operator in the context of slow environment assumption, i.e., the parallel composition of two deterministic TFSMs can have the nondeterministic behavior. Some examples can be found in [10]; however, in that paper, only the proof sketches were proposed and it was difficult to say whether more such situations can occur. We also consider some TFSM classes for which it is not the case, i.e., TFSM classes which are closed under the parallel composition. The definition of the parallel composition operator can also be used for checking whether there are live-locks in the composition of deterministic TFSMs, i.e., can be used for the verification of TFSM compositions.

The rest of the paper is organized as follows. Section 2 contains the preliminaries for classical and timed FSMs. In Section 3, the parallel composition operator is defined. In section 4, TFSM classes closed under the parallel composition are considered. Section 5 concludes the paper..

2. Preliminaries

Finite State Machines [1, 2] are widely used for modeling reactive systems that move from state to state under input stimuli and produce a predefined output response. Formally, an initialized FSM is a 5-tuple $\mathsf{S} = (S, s_0, I, O, h_S)$ where S is a finite non-empty set of states with the designated initial state s_0, I and O are input and output alphabets, and $h_S \subseteq (S \times I \times O \times S)$ is the transition (behavior) relation. A transition (s, i, o, s') describes the situation when FSM S moves from current state s to state s' when an input i is applied producing the output (response) o. The FSM S is *nondeterministic* if for some pair $(s, i) \in S \times I$, there exist several pairs $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_S$; otherwise, the FSM is *deterministic*. The FSM S is *complete* if for each pair $(s, i) \in S \times I$ there exists $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_S$; otherwise, the FSM is *partial*. FSM S is *observable* if for every two transitions (s, i, o, s_1), $(s, i, o, s_2) \in h_S$ it holds that $s_1 = s_2$.

A timed FSM can be annotated with a timed variable, timed guards and output delays [5, 6, 7]. In this paper, an initialized TFSM is a 5-tuple $\mathsf{S} = (S, s_0, I, O, \lambda_S)$ where S is a finite non-empty set of states with the designated initial state s_0, I and O are input and output alphabets, $\lambda_S \subseteq S \times I \times O \times S \times \Pi \times Z$ is the *transition relation* where the set Π is a set of *input timed guards* and Z is the set of output delays which are non-negative integers. An input timed guard $g \in \Pi$ describes the time domain when a transition can be executed and is given as an interval $<min, max>$ from $[0; \infty)$, where $< \in \{(, [\}$, $> \in \{),]\}$. The transition $(s, i, o, s', g, d) \in S \times I \times O \times S \times \Pi \times Z$ means that TFSM S being at state s accepts an input i applied at time $t \in g$ measured from the initial moment or from the moment when TFSM S produced the last output; the clock then is set to zero and S produces output o after d time units. After producing the output the clock also is set to zero.

The TFSM S is a *complete* if the union of all input timed guards at any state s under every input i equals $[0; \infty)$. The TFSM S is a *deterministic* TFSM if for each two transitions $(s, i, o_1, s_1, g_1, d_1)$, $(s, i, o_2, s_2, g_2, d_2) \in h_S$, $s_1 \neq s_2$, $d_1 \neq d_2$ or $o_1 \neq o_2$, it holds that $g_1 \cap g_2 = \varnothing$, otherwise, TFSM S is *nondeterministic*. The TFSM S is *observable* if for every two transitions (s, i, o, s_1, g_1, d), $(s, i, o, s_2, g_2, d) \in \lambda_S$ such that $g_1 \cap g_2 \neq \varnothing$ it holds that $s_1 = s_2$. In this paper, we consider a system of interacting complete deterministic TFSM components and check whether the parallel composition can have live-locks and/or the nondeterministic behavior. A *timed input* is a pair (i, t) where $i \in I$ and t is a real; a timed input (i, t) means that input i is applied to the TFSM at time instance t. A timed output is a pair (o, d) where $o \in O$ and d is the output delay for producing the output o after an input was applied. A sequence of timed inputs $\alpha = (i_1, t_1) \dots (i_n, t_n)$ is a *timed input sequence,* a sequence of timed outputs $\gamma = (o_1, d_1) \dots (o_n, d_n)$ is a *timed output sequence*.

In this work, we also use the notion of a *Finite Automaton* that is a 4-tuple $\mathsf{S} = (S, s_0, A, h_S)$, where S is a finite non-empty set of states with the designated initial state s_0, A is a finite non-empty set of actions and $h_S \subseteq (S \times A \times S)$ is the transition (behavior) relation. A transition $(s, a,$

s′) describes the situation when automaton S moves from state *s* to state *s*′ under action *a*. A trace of automaton S at state *s* is a sequence of actions which takes S from state *s* to state *s*′.

3. Defining the parallel composition operator

In this section, we propose an algorithm for deriving the parallel composition of complete deterministic TFSMs. The parallel composition operator takes two TFSMs; the input (output) alphabet of each of them is divided into two subsets of external and internal inputs (outputs). We further assume that I_S and V (I_P and U) are external and internal input alphabets of component TFSM S (or P correspondingly) while O_S and U (O_P and V) are external and internal output alphabets of component TFSM S (or P correspondingly). When an external timed input is applied to the one of components, for example, to the component TFSM S, the component produces a corresponding output response that can be an internal or an external output after appropriate number of time units. If an external output is produced then the composition is ready to accept the next external input, while the value of a time variable of another component P is increased by the time that was needed for S to communicate with the environment. If S produces an internal output then the P processes it according to its description and produces an external output to the environment or an internal input that is applied to the S. The dialog holds until an external output is produced by one of the components. If the component TFSMs fall into the infinite dialogue then the composition is said to have a *live-lock* and the behavior of the composition is not defined for a corresponding input. As usual, we assume that the system of interaction TFSMs works in the slow environment, i.e., the next external input can be applied to the composition only when the latter has produced an external output to the previous external input.

Consider the composition of TFSMs S and P in Figure 1 and an external input sequence (i_1, 0.4).(i_2, 0.5) where i_1 and i_2 are external inputs of component TFSMs S and P, respectively. Input (i_1, 0.4) is applied to component TFSM S which produces the external output (o_{11}, 1) without any dialog with component TFSM P and resets the value of its timed variable to zero. At the same time, the value of the timed variable of component TFSM P is increased to 1.4, i.e., to the time value when component TFSM S was communicating with the environment, and reaches the value 1.9 when the next external input i_2 is applied. Correspondingly, the P produces an internal output (v, 2) to the component S and the latter produces an external output (o_{12}, 3). Therefore, for the next input (i_2, 0.5) the composition produces an external output (o_{12}, 5) where five is the sum of time units needed to execute both transitions. By direct inspection a reader can assure that if the second external input i_2 were applied at time 0.9 instead 0.5 then P would produce an external output (o_{22}, 1) without a dialog with component TFSM S.

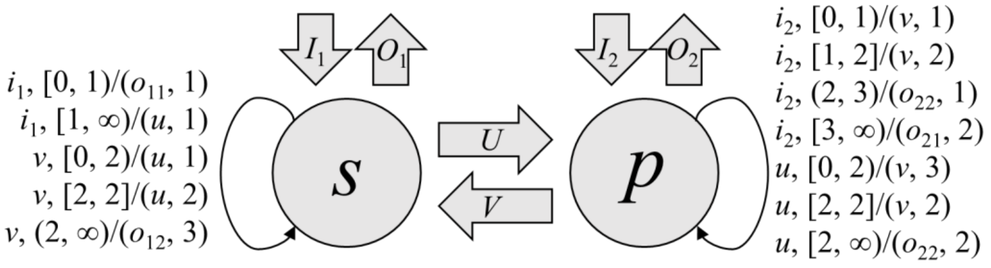


Fig. 1. Parallel composition of TFSMs S *and* P

When talking about the parallel composition over TFSMs, we would like to describe the behavior using the same model, i.e., the model of an FSM with timed guards. As a minimal timed guard has the duration one, the above example clearly shows that the parallel composition of deterministic TFSMs can have the nondeterministic behavior.

Another problem when describing the behavior of interacting deterministic FSMs which communicate in a dialogue mode, are live-locks when component FSMs fall into an infinite dialogue without producing an external output. Therefore, even under the assumption of the slow environment the parallel composition of two deterministic complete TFSMs can be partial and nondeterministic. Below we propose an algorithm for deriving the parallel composition of two FSMs with timed guards. In general case, this composition can have live-locks and the nondeterministic behavior; both features can be detected during the parallel composition construction.

Parallel composition operator:

Given a positive integer B, we use the following notations. The set G_B of intervals is the set $\{[a, a], (a, a + 1), [B, B], (B, \infty): a \in N \text{ and } a < B\}$, where N is the set of non-negative integers. Given two intervals $g, g' \in G_B$, we define the composition $g \circ g'$ of these intervals. If $g = [a, a]$ and $g' = [b, b]$ then $g \circ g' = [a + b, a + b]$; if $g = (a, b)$ and $g' = (a', b')$ then $g \circ g' = (a + a', b + b')$ assuming that $a + \infty = \infty$ for any integer a.

For each interval g, we specify a corresponding set $F_B(g)$ of G_B items. $F_B(g) = \{g' : g' \in G_B, g' \cap g \neq \varnothing\}$.

The behavior of a TFSM can be adequately described using a classical FSM that is called the *FSM abstraction* of the TFSM [11]. Given a complete observable possibly deterministic TFSM $\mathsf{S} = (S, s_0, I, O, \lambda_\mathsf{S})$ with the largest finite boundary of timed guards B_S and the largest output delay D_S, the behavior of a corresponding FSM abstraction $\mathsf{A}_\mathsf{S}(B) = (S, s_0, I_A, O_A, \lambda_{\mathsf{S}A})$, where $B \geq B_\mathsf{S}$, $I_A = I \times G_B$, $O_A = O \times \{0, 1, \ldots, D_\mathsf{S}\}$, can be described as follows. There is a transition $(s, (i, g_i), (o, d), s') \in$

$\lambda_{\mathsf{S}A}$ if and only if there is a transition $(s, i, o, s', g, d) \in \lambda_{\mathsf{S}}$ such that $g_i \subseteq g$. For deterministic TFSMs S and P in Figure 1 corresponding FSM abstractions are shown in Figure 2.

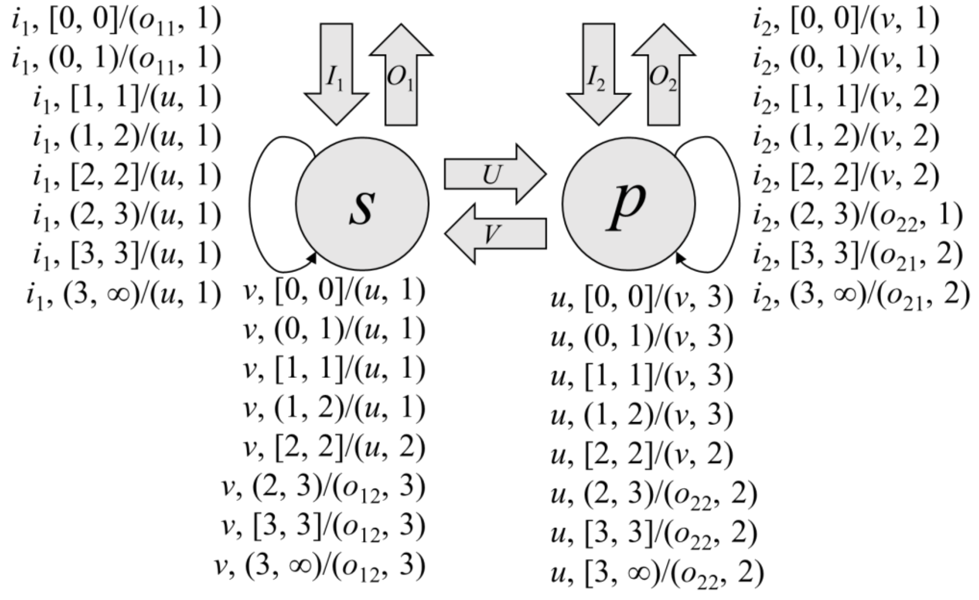


Fig. 2. FSM abstractions of TFSMs S and P

Algorithm for deriving the parallel composition of initialized deterministic complete FSMs with timed guards S and P

Input. Initialized deterministic complete FSMs with timed guards S = $(S, s_0, I_{\mathsf{S}} = I_s \cup V, O_{\mathsf{S}} = O_s \cup U, h_{\mathsf{S}})$ and P = $(P, p_0, I_{\mathsf{P}} = I_p \cup U, O_{\mathsf{P}} = O_s \cup V, h_{\mathsf{P}})$, where I_s and O_s (I_p and O_p) are external input and output alphabets of the component TFSMs S (P) while V and U (U and V) are internal input and output alphabets of the component FSM S (P); the alphabets are pairwise disjoint

Output. Parallel composition of S and P that is a FSM T with timed guards or one of messages «The system has a live-lock» or «The behavior of the system is nondeterministic»

Step 1. Derive the global automaton Q = $(Q, \{(i, g)/(r, d): i \in I_s \cup I_p, o \in O_s \cup O_p, g \in G_B, d \in \mathrm{Z}\}, q_0, \lambda_{\mathsf{Q}})$, where B is maximum of B_{S} and B_{P} and each state $q \in Q$ that is a *stable* state, is a 4-tuple $(s, g_s, p, g_p) \in S \times G_B \times P \times G_B$; each state $q \in Q$ that is an *unstable* state, is a 5-tuple $(s, g_s, p, g_p, (a, g))$ where $(s, g_s, p, g_p) \in S \times G_B \times P \times G_B$ and $a \in U \cup V, g \in G_B$. By definition, the initial state $q_0 = (s_0, [0, 0], p_0, [0, 0])$ is stable. A state is stable if and only if every incoming transition to a state is labelled by an action $(r, g)/(o, d)$ where o is the external output of S or P. At each stable state every transition under an action $(i, g)/(r, d)$, where i is the external input of S or P, is defined. At an unstable state that is a 5-tuple $(s, g_s, p, g_p, (a, g))$, the only defined transition is under an action $(a, g_s)/(o, d)$, $r \in O_{\mathsf{S}} \cup O_{\mathsf{P}}$.

Step 2. Derive FSM abstractions $\mathsf{A}_{\mathsf{S}}(B)$ and $\mathsf{A}_{\mathsf{P}}(B)$.

```
  **For each** external input $i \in I_s$, interval $g \in G_B$ and stable state $(s, [0,0], p, g_p)$ of the automaton Q and transition $(s, (i, g), (r, d), s')$ of the FSM abstraction $A_S(B)$
    **If** $r$ is internal **then** add to Q transitions $((s, [0,0], p, g_p), (i, g)/(r, d), (s', [0,0], p, g_p', (r, [d, d]))$ where $g_p' \in F_B(g \circ g_p \circ [d, d])$;
    **Else** add to Q transitions $((s, [0,0], p, g_p), (i, g)/(r, d), (s', [0,0], p, g_p'))$, where $g_p' \in F_B(g \circ g_p \circ [d, d])$;
  **For each** external input $i \in I_p$, interval $g \in G_B$ and stable state $q = (s, g_s, p, [0,0])$ of the automaton Q and transition $(p, (i, g), (r, d), p')$ of the FSM abstraction $A_P(B)$
    **If** $r$ is internal **then** add to Q transitions $((s, g_s, p, [0,0]), (i, g)/(r, d), (s, g_s', p', [0,0], (r, [d, d]))$ where $g_s' \in F_B(g \circ g_s \circ [d, d])$;
    **Else** add to Q transitions $((s, g_s, p, [0,0]), (i, g_i)/(r, d), (s, g_s', p', [0,0])$, where $g_s' \in F_B(g \circ g_s \circ [d, d]$;
  **For each** unstable state $(s, [0,0], p, g_p, (r, [d, d])$ of global automaton Q with an incoming transition labeled by $(i, g)/(r, [d, d])$ determine a transition $(p, (r, g_p), (m, k), p')$ of the FSM abstraction $A_P(B)$;
    **If** $m$ is internal **then** add to Q a transition $((s, [0,0], p, g_p, (r, [d, d])), (r, [d, d])/(m, k), (s, [k, k], p', [0, 0], (m, [k, k]))$;
    **Else** add to Q a transition $((s, [0,0], p, g_p, (r, [d, d])), (r, [d, d])/(m, k), (s, [k, k], p', [0, 0]))$
  **For each** unstable state $(s, g_s, p, [0,0], (r, [d, d]))$ of global automaton Q with an incoming transition labeled by $(i, g)/(r, [d, d])$ determine a transition $(s, (r, g_s), (m, k), s')$ of the FSM abstraction $A_P(B)$;
    **If** $m$ is internal **then** add to Q a transition $(s, g_s, p, [0,0], (r, [d, d])), (r, [d, d])/(m, k), (s', [0, 0], p, [k, k], (m, [k, k]))$
    **Else** add to Q a transition $((s, g_s, p, [0,0], (r, [d, d])), (r, [d, d])/(m, k), (s', [0, 0], p, [k, k]))$
**Step 3.** Derive the parallel composition of S and P in the form of FSM abstraction $A_T$
  **If** the global automaton Q has a cycle where all transitions are labelled with internal actions **then** return the message «The composition has a live-lock»;
  **Else** Derive an FSM $A_T$:
  -   states of $A_T$ correspond to stable states of the global automaton Q;
  -   for each trace $\gamma = (i_1, g_1)/(o_1, d_1), \dots, (i_n, g_n)/(o_n, d_n)$ which takes Q from a stable state $q_1$ to stable state $q_2$, add to $A_T$ a transition $(q_1, (i_1, g_1), (o_n, d), q_2)$ where $d = d_1 + \dots + d_n$ is the sum of all output delays in trace $\gamma$;
  -   derive the observable form of FSM $A_T$ [12];
```

If A_T is nondeterministic **then** return the message «The composition behavior cannot be described by a deterministic TFSM».
Else the FSM abstraction A_T is translated into TFSM T.

Consider as an example of parallel composition of TFSMs S and P in Figure 1. A fragment of the global automaton derived at Step 2 of the above algorithm and a fragment of the corresponding FSM A_T are shown in Figure 3.

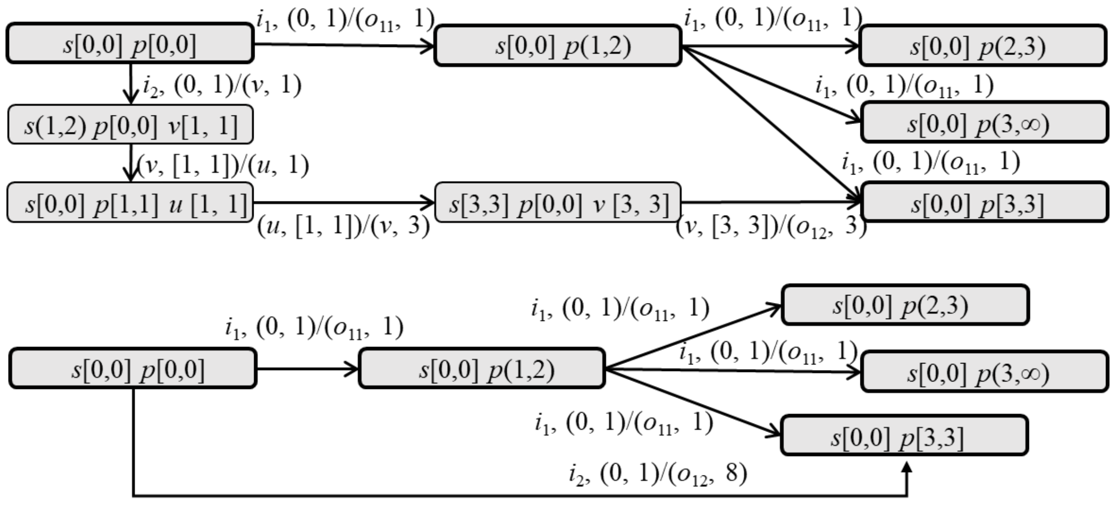


Fig. 3. A fragment of the global automaton Q for TFSMs S and P and a fragment of the corresponding FSM A_T

4. Selected TFSM classes closed under the parallel composition operator

When analyzing Algorithm 1, we conclude that the derived composed machine can be nondeterministic when at least one stable state of global automaton has interval (a, b). In this case, the response of TFSM components to input $(i, (a', b')$ cannot be uniquely specified. In this section, we consider special classes of TFSM composition for which the composition behavior is deterministic.

We first note that the global automaton Q and the corresponding composed machine are deterministic if $G_B = \{[0, 0], [1, 1], \ldots, [B, B], (B, \infty)\}$. Another case of deterministic behavior occurs when for each external input every component produces an internal output, as the automaton Q is then taken to an unstable state from which there the only deterministic transition is defined.

Below we formally specify deterministic TFSM classes which are closed under the paper composition operator. Differently from [10], the proofs to Propositions 1 and 2 are direct corollaries to the definition of the parallel composition operator.

Proposition 1 [10]. Let S and P are deterministic complete FSMs with timed guards. Parallel composition of TFSMs S and P is a deterministic TFSM if for each transition $(q, i, a, q', g, d) \in h_S \cup h_P$, where i is an external input, it holds that a is an internal output.

Proposition 2. Let S and P are deterministic complete FSMs with timed guards. Parallel composition of TFSMs S and P is deterministic TFSM if external inputs can be applied to TFSM composition only at integer time instances.

In both above cases, the value of the clock variable of each component is an integer when an external output is produced and $G_B = \{[0, 0], [1, 1], \ldots, [B, B], (B, \infty)\}$.

Proposition 3. Let S and P be deterministic complete FSMs with timed guards. The parallel composition of TFSMs S and P is a deterministic TFSM if the minimum output delay in TFSM S and P is bigger than the maximum of B_S and B_P.

In these cases, the value of the clock variable of each component is 0 or (B, ∞) when an external output is produced, respectively $G_B = \{[0, 0], (B, \infty)\}$.

Similar to parallel composition operator over classical FSMs [2], the parallel composition of FSMs with timed guards is a complete TFSM if the system of communicating TFSMs has no live-locks.

5. Conclusion

In this paper, we propose an algorithm for deriving a composed machine for a system of communicating FSMs with timed guards, namely, we define the parallel composition operator for a pair of complete deterministic FSMs with timed guards. Similar to classical finite state machines, we assume the slow environment, i.e., the next external input is applied to the system only when the system has produced an external output to the previous external input. However, differently from classical finite state machines, the set of deterministic TFSMs is not closed under the parallel composition operator. When deriving the parallel composition of two communicating FSMs with timed guards, we also detect live-locks in the system (if any) and the nondeterministic behavior if the latter occurs. We also specify some TFSM classes such that there is no nondeterministic behavior when TFSM components of these classes are composed.

Acknowledgement

This work is partly supported by RFBR project № 19-07-00327/19.

References

1. Villa T., Kam T., Brayton R.K., Sandgiovanni-Vincentelli A. Synthesis of Finite State machines: Logic Optimization, 1997. 520 p.
2. Villa T., Yevtushenko N., Brayton R.K., Mishchenko A., Petrenko A., Sangiovanni-Vincentelli A. The Unknown Component Problem. Theory and Applications. Springer, 2012. 312 p.
3. Chow T.S., Test design modeled by finite-state machines // IEEE Trans. Software Eng. 1978. Vol. 4. No. 3. P. 178–187.
4. Alur R., Dill D.L. A theory of timed automata // Theoretical Computer Science. 1994. Vol. 126. P. 183–235.
5. Krichen M., Tripakis S., Conformance testing for real-time systems // Formal Methods Syst. Des. 2009. Vol. 34. No. 3, P. 238–304.
6. Merayo M.G., Nunez M., Rodriguez, I. Formal testing from timed finite state machines // Comput. Networks: Int. J. Comput. Telecommun. Networking. 2008. Vol. 52. No. 2. P. 432–460.
7. Bresolin D., El-Fakih K., Villa T., Yevtushenko N. Deterministic timed finite state machines: Equivalence checking and expressive power // Int. Conf. GANDALF. 2014. P. 203–216.
8. Springintveld J., Vaandrager F., D’Argenio P. Testing timed automata // Theor. Comput. Sci. 2001. Vol. 254. No. 1–2, P. 225–257.
9. Kondratyeva O., Yevtushenko N., Cavalli A. Solving parallel equations for Finite State Machines with Timeouts // Trudy ISP RAN/Proc. ISP RAS,2014. Vol. 26, No. 6. P. 85–98.
10. Tvardovskii A.S., Laputenko A.V. On the possibilities of FSM description of parallel composition of timed Finite State Machines // Trudy ISP RAN/Proc. ISP RAS, 2018. Vol. 30. No. 1. P. 25-40 (in Russian). DOI: 10.15514/ISPRAS-2018-30(1)-2
11. Bresolin D., Tvardovskii A., Yevtushenko N., Villa T., Gromov M. Minimizing Deterministic Timed Finite State Machines // IFAC-PapersOnline. 2018. Vol. 51. No. 7. P. 486–492.
12. Starke P. Abstract Automata / P. Starke // American Elsevier, 1972. 419 p.