UDK 004.415.52

Requirements patterns in deductive verification of process-oriented programs and examples of their use

Chernenko I. M. (Institute of Automation and Electrometry SB RAS)

Process-oriented programming is a promising approach to the development of control software. Control software often has high reliability requirements. Formal verification methods, in particular deductive verification, are used to prove the correctness of such programs regarding the requirements. Previously, a temporal requirements language [2] was developed to specify temporal requirements for deductive verification of process-oriented programs. It was also shown that a significant part of the requirements falls into a small number of classes. Requirements patterns was developed for these classes. In this paper, we present a collection of process-oriented programs and requirements for them. Requirements are formalized in the temporal requirements language and classified according the set of patterns. We also define a new requirement pattern. These results can be used in the research of formal verification methods for process-oriented programs, in particular in the research of methods of proving verification conditions.

Keywords: deductive verification, temporal requirements, control software, process-oriented programs

1. Introduction

Formal verification is a crucial part in the development of safety-critical software, in particular, control software. Deductive verification is one of formal verification methods in which program properties are formalized in the form of logical formulas expressing relations between program variables (preconditions, postconditions and invariants) and added to the programs as annotations. Then for this annotated program, the generation and proving of verification conditions are performed.

Process-oriented programming [11] is one of approaches to control software development. A process-oriented program is defined as a sequence of interacting processes which is executed in the control loop. Each process is represented by a set of named executable codes called process states.

Control software often has temporal requirements. Previously, we developed a deductive verification approach [1] for process-oriented programs with temporal requirements. This approach entails utilizing control loop invariants and storing the history of changes in variable values to specify temporal requirements. In our work [2], we introduced the DV-TRL annotation language oriented to deductive verification for formalizing requirements to process-oriented programs. We also formalized a set of 45 requirements for 10 case studies and discovered that a considerable portion of these requirements can be classified into a few distinct classes. Based on these findings, we established four requirement patterns.

To verify temporal requirements, model checking is a commonly used method where temporal properties are specified using temporal logic formulas (e.g., LTL or CTL). Model checking tools then automatically check if the program model satisfies the specified requirements. However, one limitation of model checking is the state explosion problem. Hence, deductive verification is also employed to verify temporal requirements.

Deductive verification is typically used to verify functional requirements [5, 6]. In deductive verification, requirements are described by assertions at certain points in the program linking the current values of variables at these points [4]. Nevertheless, deductive verification is also used to verify temporal requirements. For example, in STeP [8], verification rules are used to reduce the proof of correctness of a program in the SPL language with respect to a temporal formula to a set of verification conditions that are formulas of first-order logic. In [7] the deductive verification of control software with temporal properties specified using timing charts is presented. The Why3 system is used for verification. Events and stable states of the timing chart are represented in Why3 by loops, the body of which corresponds to the iteration of the control loop, and the guard specifies the stable state of the timing chart. In our work, we define temporal properties in the form of invariants of the control loop. Unlike STeP, we do not develop special verification rules for temporal formulas, but use the rules of axiomatic semantics, similar to the rules of Hoare logic.

Temporal specification of requirements plays a crucial role in the development of critical control software. However, this process is often burdensome and prone to errors. To address these challenges, research is being undertaken to simplify the temporal specification. For instance, in [9], the authors introduce a classification encompassing commonly encountered temporal requirements in specification tasks. Similarly, we have developed our own classification and established patterns. While our classification may be incomplete, it incorporates requirement classes identified in real control system specifications.

This paper aims to expand the set of temporal requirements and establish control programs in the process-oriented language poST [12]. Additionally, we enhance the collection of requirement patterns to align with the extended set of requirements.

The rest of the paper has the following structure. We describe the temporal requirements language in Section 2 and requirement classification in Section 3. In Section 4, we present the set of case studies.

This work is supported by the Russian Ministry of Education and Science, project no. 122031600173-8.

2. Temporal Requirements Language DV-TRL

The temporal requirement language DV-TRL introduced in [2] is based on *state* date type storing the history of all changes of program variables and the set of specialized functions over this data type. The *state* data type is defined by the following set of constructors:

- $emptyState : state$ the initial empty state of the control system;
- $toEnv : state \to state$ inputs to the environment
- $setVar : state \times variable \times value \to state$ sets values to variables;
- $setPstate : setPstate : state \times process \times pstate \to state$ sets process states;
- $reset : state \times process \to state$ resets local clocks of processes.

A corresponding constructor is defined for each type of changes in programs.

Following domain-specific functions over the *state* data type are used in program annotations:

- $getVar : state \times variable \to value$ returns values of variables;
- $getPstate : state \times process \to pstate$ returns current states of processes;
- $substate : state \times state \to bool$ checks that the first state is a substate of the second state. A state s' is a substate of s if $s' = s$, or there exist a state s', a constructor c, and values v_1, ..., v_n such that $s' = c(s'', v_1, \ldots, v_n)$, and s'' is a substate of s;
- $toEnvNum : state \times state \to nat$ returns a number of application of constructor $toEnv$ for getting the second state from the first substate. The score starts anew if the reset constructor meets;
- $toEnvP : state \to bool$ checks whether the state has the form $toEnv(s)$ for some s;

These functions allow describing the requirements for process-oriented programs in a natural way.

Here the *value* type is a union of types *bool*, *nat* and *int*. Types *bool*, *nat* and *int* describe sets of logical constants (true and false), natural numbers and integers numbers, respectively.

The *variable*, *process*, and *pstate* types are used to encode the names of variables, processes, and process states of a process-oriented program respectively.

3. Requirements Patterns

In this section, we will be discussing 5 temporal requirements patterns. These patterns are formulated as parametric formulas in the DV-TRL language. The initial 4 patterns were previously introduced in [2]. We add one more pattern and present the formula only for it.

The first pattern establishes requirements that specify that event E_2 should occur no later than τ after event E_1. An example of this is the first requirement for the turnstile control program, which we will discuss further below.

The second pattern describes requirements that state that a specific event will occur after one iteration of the loop. An example of this is the first requirement for the thermopot control program, which we will presented on later.

The third pattern describes requirements that assert that events should occur at one specific point within the control loop. An example of this is the sixth requirement for the revolving door control program, which we will examine in more detail below.

The fourth pattern for requirements combines the first and second patterns. If event E_1 occurs, event E_2 must occur within a time interval of τ. In this scenario, the formula that describes E_1 imposes restrictions on the variable values in two different states, separated by one iteration of the control loop. An example of this class is the first requirement for the pedestrian crossing light control program that will be discussed later.

The fifth pattern describes requirements that state that if event E_1 occurs, event E_2 can only occur after a minimum time interval of τ. The formula that describes event E_1 imposes restrictions on variable values in two distinct states, with the time between them being one iteration of the control loop. The requirements pattern for this class can be expressed as follows:

E_1 has happened, then event E_2 does not happen for at least τ.

$$
\begin{aligned}
&p_5(s, \tau, vc_1, vc_2) \equiv\\
&\text{toEnvP}\, s \wedge\\
&(\forall s_1 s_2 s_3.\, \text{substate}\, s_1\, s_2 \wedge \text{substate}\, s_2\, s_3 \wedge \text{substate}\, s_3\, s \wedge \text{toEnvP}\, s_1 \wedge \text{toEnvP}\, s_2 \wedge\\
&\text{toEnvP}\, s_3 \wedge \text{toEnvNum}\, s_1\, s_2 = 1 \wedge \text{toEnvNum}\, s_2\, s_3 < \tau \wedge vc_1(s_1, s_2) \longrightarrow vc_2(s_3),
\end{aligned}
$$

where variable constraint vc_1 describes the event E_1, vc_2 describes the event E_2. This class includes, for example, the fourth requirement for the fridge control program considered below.

4. Case Studies

This section presents the set of case studies. We provide descriptions of the case studies and requirements for them as well as classify these requirements according to the patterns and provide examples of their formalization. The poST programs can be found on GitHub [13].

4.1. Turnstile

In this case study, we examine a turnstile as a controlled device. The turnstile is equipped with a coin acceptor, doors operated by a signal (open), an LED (enter) indicating the possibility of passage, and two sensors to detect the presence of a user (PdOut) and the opening of the doors (opened). The doors remain locked until a payment signal (paid) is received from the coin acceptor, after which they open. If the user does not pass through within 10 seconds after the doors open, they will automatically close. After a successful payment, the coin acceptor is locked. The coin acceptor is unlocked by a reset signal once the turnstile is closed.

For the turnstile control program, we propose the following 7 requirements:

1. The open signal should remain true for a maximum of 10 seconds.
2. If the turnstile has been closed and the payment has not been made, it will not open until the payment is made.
3. After receiving the paid signal from the coin acceptor, the open signal should be given immediately.
4. After receiving the PdOut signal indicating the passage of a user, the turnstile must be closed within a maximum of 1 second.
5. The open signal should remain true for a minimum duration of 1 second.
6. After the opened signal appears and until it is reset, the enter LED should be lit.
7. After the turnstile is closed, the signal reset should be given to unlock the coin acceptor.

The requirement 1 belongs to the 1st class. The requirements 2, 3 and 7 belong to the 2nd class. The requirement 4 belongs to the 4th class. The requirement 5 belong to the 5th class. The requirement 6 belong to the 3rd class. For example, the first requirement is formalized as the following annotation:

$$\mathrm{toEnvP}\, s \wedge$$
$$(\forall s1.\, \mathrm{substate}\, s1\, s \wedge \mathrm{toEnvP}\, s1 \wedge \mathrm{toEnvNum}\, s1\, s \geq 100 \wedge \mathrm{getVarBool}\, s1\, open' \longrightarrow$$

$(\exists s3.\, \mathrm{toEnvP}\, s3 \wedge \mathrm{substate}\, s1\, s3 \wedge \mathrm{substate}\, s3\, s \wedge$

$\mathrm{toEnvNum}\, s1\, s3 \leq 100 \wedge \neg\, \mathrm{getVarBool}\, s3\, open' \wedge$

$(\forall s2.\, \mathrm{toEnvP}\, s2 \wedge \mathrm{substate}\, s1\, s2 \wedge \mathrm{substate}\, s2\, s3 \wedge s2 \neq s3 \longrightarrow \mathrm{getVarBool}\, s2\, open')))$

4.2. Pedestrian Crossing Light

In this case study, a pedestrian crossing light with a button installed at the crossing is considered as a controlled device. Its regular state is "red". When a pedestrian appears at the crossing, he presses the button. If the green signal of the pedestrian crossing light was on more than 10 seconds ago, then the green signal will turn on 5 seconds after pressing the button. If the green light was on no more than 10 seconds ago, then the green will turn on after the timeout between transitions equal to 15 seconds. If green is turned on, it will be on for 30 seconds, then red turns on.

Thus, there is one input signal ("the button is pressed") and one control signal ("green is on"). The program gets the input signal and, depending on it, controls the traffic light signal.

This program should satisfy the following requirements:

1. If the red light is on and the button is pressed, the green light will be activated within a maximum of Tr seconds.
2. If the green light just has turned on, then the green light will be on for at least Tg seconds.
3. Once the green light has just been activated, it will transition to red within a maximum of Tg seconds.
4. If the red light just has turned on, then the red light will be on for at least Tr seconds.

Here Tr is the maximum time during which a pedestrian waits for the green signal of the traffic light to turn on after pressing the button, equal to 15 seconds, Tg is the duration of the green signal of the traffic light, equal to 30 seconds.

The requirements 1 and 3 belong to the 4th class. The requirements 2 and 4 belong to the 5th class. For example, the first requirement is formalized as the following annotation:

$\mathrm{toEnvP}\, s \wedge$

$(\forall s1 s2.\, \mathrm{substate}\, s1\, s2 \wedge \mathrm{substate}\, s2\, s \wedge \mathrm{toEnvP}\, s1 \wedge \mathrm{toEnvP}\, s2 \wedge \mathrm{toEnvNum}\, s1\, s2 = 1 \wedge$

$\mathrm{toEnvNum}\, s2\, s \geq Tr \wedge \mathrm{getVarBool}\, s1\, trafficLight = RED \wedge$

$\mathrm{getVarBool}\, s1\, requestButton = NOT\_PRESSED \wedge$

$\mathrm{getVarBool}\, s2\, requestButton = PRESSED \longrightarrow$

$(\exists s4.\, \mathrm{toEnvP}\, s4 \wedge \mathrm{substate}\, s2\, s4 \wedge \mathrm{substate}\, s4\, s \wedge \mathrm{toEnvNum}\, s2\, s4 \leq Tr \wedge$

$$\text{getVarBool}\, s4\, trafficLight = GREEN \wedge$$
$$(\forall s3.\, \text{toEnvP}\, s3 \wedge \text{substate}\, s2\, s3 \wedge \text{substate}\, s3\, s4 \wedge s3 \neq s4 \longrightarrow$$
$$\text{getVarBool}\, s3\, trafficLight = RED)))$$

4.3. Revolving Door

In this case study, a revolving door is considered as a controlled device. Revolving doors are installed at the entrances to buildings with a large flow of visitors. The device consists of a three- or four-section door rotating around a vertical axis, a motor and a brake for instant stop of the door.

In the absence of users, the door is stationary, and when the user approaches, it begins to rotate. The rotation continues while the user is inside the rotation space. The approach of the user and his presence inside the rotation space is registered by the motion sensor (user). If the user leaves the rotation space, then after a certain time the rotation stops.

The pressure sensor registers the pressure on the sectional partitions. Rotation is suspended for a short time when pressure is exerted to the partitions.

We offer the following requirements for this program:

1. When a user enters, the door starts to rotate if pressure is not detected.
2. Rotation continues while the user is inside the rotation space if pressure is not detected.
3. If the user has left the rotation space, then the rotation should stop after no more than 1 second if users do not reappear during this time.
4. If pressure is detected, then rotation should be suspended for at least 1 second.
5. If pressure is no longer detected then rotation should resume no more than 1 second.
6. Simultaneous appearance of rotation and brake signals is prohibited.

Here the requirements 1 and 2 belong to the 2nd class. The requirements 3 and 5 belong to the 4th class. The requirement 4 belongs to the 5th class. The requirement 6 belongs to the 3rd class. For example, the sixth requirement is formalized as the following annotation:

$$\text{toEnvP}\, s \wedge$$
$$(\forall s1.\, \text{substate}\, s1\, s \wedge \text{toEnvP}\, s1 \wedge$$
$$\text{getVarBool}\, s1\, brake = True \longrightarrow \text{getVarBool}\, s2\, rotation = False)$$

4.4. Fridge

In this case study, a fridge is considered as a controlled device. The fridge consists of a fridge and a freezer and has two compresses. The temperature in the fridge is registered by the fridgeTempGreaterMin and fridgeTempGreaterMax sensors, showing whether the temperature exceeds the minimum and maximum values, respectively. To control the temperature in the freezer, the device has sensors freezerTempGreaterMin and freezerTempGreaterMax. The fridge maintains the temperature in the range between the minimum and maximum values. When the temperature in the fridge is exceeded, the compressor (fridgeCompressor) turns on, which turns off when the temperature reaches the minimum value. The freezer Compressor is used for the freezer. When the fridge door is opened, the lighting turns on, which turns off when it is closed. If the fridge door is open for more than 30 seconds, a sound signal (dorsignal) is given.

We offer the following requirements for this program:

1. When the fridge door is opened, the lighting turns on.
2. When the fridge door is closed, the lighting turns off.
3. If the fridge door is open, the signal is given after no more than 30 seconds if the user does not close the door during this time.
4. The sound signal is not given spontaneously. The signal is given only if the door is open for at least 30 seconds.
5. If the temperature in the fridge exceeds the maximum, the compressor turns on.

Here the requirements 1, 2 and 5 belong to the 2nd class. The requirement 3 belongs to the 1st class. The requirement 4 belongs to the 5th class. For example, the fourth requirement is formalized as the following annotation:

$$\begin{array}{l}
\text{toEnvP}\, s \wedge \\
(\forall s1 s2 s3.\, \text{substate}\, s1\, s2 \wedge \text{substate}\, s2\, s3 \wedge \text{substate}\, s3\, s \wedge \text{toEnvP}\, s1 \wedge \text{toEnvP}\, s2 \wedge \\
\text{toEnvP}\, s3 \wedge \text{toEnvNum}\, s1\, s2 = 1 \wedge \text{toEnvNum}\, s2\, s3 < OPEN\_DOOR\_TIME\_LIMIT \\
\text{getVarBool}\, s1\, fridgeDoor = CLOSED' \wedge \text{getVarBool}\, s2\, fridgeDoor = OPEN \longrightarrow \\
\neg\, \text{getVarBool}\, s3\, doorSignal)
\end{array}$$

4.5. Thermopot

In this task, a thermopot is considered as a controlled device. A thermopot is the device that combines the functions of a kettle and a thermos. The thermopot has three temperature modes. It heats the water to the temperature corresponding to the selected temperature mode (selectedTemp), and maintains this temperature. The device contains a housing with a sealed flask, which allows it to maintain the required temperature for a long time, a lid and a heating element (heater). There is a control panel with three buttons (button1, button2, button3) on the lid that allow you to select the desired temperature mode. The boiling button (boilingButton) is used to turn on the boiling. During boiling, the lid is locked. Output signal lid controls the lid locking. After boiling, the thermopot switches to the temperature maintenance mode. In the temperature maintenance mode, the heating element turns on when the water temperature becomes more than 5 degrees less than the set temperature. The boilingMode and maintainingMode indicators show whether the thermopot is in the boiling and temperature maintenance mode, respectively.

We propose the following requirements for this program:

1. Until the required temperature is reached, the lid is locked.
2. When the set temperature is reached, the heating element turns off.
3. The heating element turns on when the water temperature becomes more than 5 degrees less than the set temperature.
4. When one of the temperature mode selection buttons is pressed, the corresponding required temperature is set.

   item If the boiling button is not pressed, the heating will not turn on.

Here the requirements 1, 2, 3 and 5 belong to the 2nd class. The requirement 4 belongs to the 3rd class. For example, the first requirement is formalized as the following annotation:

$$\begin{aligned}&\text{toEnvP}\, s \wedge\\ &(\forall s1 s2.\, \text{substate}\, s1\, s2 \wedge \text{substate}\, s2\, s \wedge \text{toEnvP}\, s1 \wedge \text{toEnvP}\, s2 \wedge \text{toEnvNum}\, s1\, s2 = 1 \wedge\\ &\text{getVarBool}\, s1\, boilingMode' \wedge \text{getVarInt}\, s2\, temperature' < BOILING_POINT' \longrightarrow\\ &\text{getVarBool}\, s2\, lid' = LOCKED')\end{aligned}$$

5. Conclusion

In this paper, we proposed a collection of control programs in process-oriented poST language. We formulated a set of temporal requirements for each program. These requirements have been classified according previously developed requirements patterns. This classification confirmed that most of the requirements belong to previously introduced classes, but some requirements form a new class. We defined the pattern for this requirements class. The developed collection of process-oriented programs can be used in the research of methods of formal verification of such programs, in particular, in the development of a methodology for proving verification conditions.

In the future, we plan to investigate the possibility of automation of proving verification conditions for requirements belonging to developed classes.

References

1. Anureev I., Garanina N., Liakh T., Rozov A., Zyubin V., Gorlatch S. Two-step deductive verification of control software using Reflex. // International Andrei Ershov Memorial Conference on Perspectives of System Informatics. Springer. 2019. P. 50–63.
2. Chernenko I.M., Anureev I.S., Garanina N.O., Staroletov S.M. A Temporal Requirements Language for Deductive Verification of Process-Oriented Programs // 2022 IEEE 23rd International Conference of Young Professionals in Electron Devices and Materials (EDM). 2022. P. 657-662.
3. Dwyer M.B., Avrunin G.S., Corbett J.C. Patterns in property specifications for finite-state verification // Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002). 1999. P. 411-420.
4. Fillitre J.C. Deductive software verification // Int J Softw Tools Technol Transfer. 2011. Vol. 13. P. 397-403.
5. Gurov D., Herber P., Schaefer I. Automated Verification of Embedded Control Software // International Symposium on Leveraging Applications of Formal Methods / Springer. 2020. P. 235-239.
6. Gurov D., Lidström C., Nyberg M., Westman J. Deductive Functional Verification of Safety-Critical Embedded C-Code: An Experience Report // Critical Systems: Formal Methods and Automated Verification. Cham : Springer International Publishing, 2017. P. 3-18.
7. Lourenço C.B., Cousineau D., Faissole F. et al. Automated Verification of Temporal Properties of Ladder Programs // Formal Methods for Industrial Critical Systems / Ed. by Lluch Lafuente A., Mavridou A. Cham : Springer International Publishing, 2021. P. 21-38.
8. Manna Z., Bjørner N.S., Browne A. et al. An Update on STeP: Deductive-Algorithmic Verification of Reactive Systems // Tool Support for System Specification, Development and Verification / Ed. by Berghammer R., Lakhnech Y. Vienna: Springer Vienna, 1999. P. 174-188.
9. Mekki A., Ghazel M., Toguyeni A. A. K. Assisting Temporal Requirement Specification // Computer Technology and Application. 2012. Vol. 3. P. 47-55.
10. Sin C. O., Kim Y.S. TimeLine Depiction: an approach to graphical notation for supporting temporal property specification // Innovations in Systems and Software Engineering. 2023. P. 319-335.
11. Zyubin V. E. Hyper-automaton: a Model of Control Algorithms // 2007 Siberian Conference on Control and Communications. 2007. P. 51-57.
12. Zyubin V.E., Rozov A.S., Anureev I.S. et al. PoST: A Process-Oriented Extension of the IEC 61131-3 Structured Text Language // IEEE Access. 2022.
13. Chernenko I. PoST verification condition generator. URL: https://github.com/ivchernenko/post_vcgenerator (online; accessed: 21.10.2023)