UDK 004.415.52

Pattern-based approach to automation of deductive verification of process-oriented programs[*]

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

Process-oriented programming is an approach to the development of control software in which a program is defined as a set of interacting processes. PoST is a process-oriented language that extends ST language from the IEC 61131-3 standard. In the field of control software development, formal verification plays an important role because of the need to ensure the high reliability of such software. Deductive verification is a formal verification method in which a program and requirements for it are presented in the form of logical formulas and logical inference is used to prove that the program satisfies the requirements. Control software is often subject to temporal requirements. We formalize such requirements for process-oriented programs in the form of control loop invariants. But control loop invariants representing requirements are not sufficient for proving program correctness. Therefore, we add extra invariants that contain auxiliary information. This paper addresses the problem of automating deductive verification of process-oriented programs. We propose an approach in which temporal requirements are specified using requirement patterns that are constructed from basic patterns. For each requirement pattern the corresponding extra invariant pattern and lemmas are defined. The proposed approach allows us to make the deductive verification of process-oriented programs more automated.

Keywords: deductive verification, temporal requirements, requirement pattern, loop invariant, control software, process-oriented programming

1. Introduction

Process-oriented programming [27] is a promising method for developing control software. This programming paradigm allows one to describe a program as a set of interacting processes. Each process is an extended finite state machine and is defined by a set of named states that contain the program code. Besides the active states defined in the code, each process has two inactive states: the normal stop state *STOP*, and the error stop state *ERROR*. Program execution follows a cyclical pattern; in each iteration of the control loop, all program processes are executed sequentially in their current states. The duration a process remains in its current state is controlled by a timeout statement. A timer is associated with each process to control

[*] This work was supported by the Russian Ministry of Education and Science, project no. 122031600173-8.

this time. The timer resets whenever the process transits to a different state, and it can also be reset programmatically. Processes have the ability to start and stop other processes, as well as to check whether another process is active or inactive. When starting, a process is in its state defined first in the program text. When the program starts, its first process starts while all subsequent processes remain in the state *STOP*.

PoST language [28] is a process-oriented language that extends ST language from the IEC 61131-3 standard [1]. A poST program consists of variable declarations and process definitions. A variable declaration contains declaration of input variables `VAR_INPUT` whose values are changed by the environment at each iteration of the control loop, output variables `VAR_OUTPUT` that define control signals or local variables `VAR`. A process definition contains a sequence of state definitions.

Control software requires formal verification because it has high reliability requirements. Deductive verification [14] is one of the formal verification methods in which requirements are formalized in the form of logical formulas, verification conditions that are logical formulas whose truth guarantees the program correctness are generated, and then the verification conditions are proved. For each loop in the program, a loop invariant must be specified that is true when entering the loop and after each its iteration.

An important class of requirements for control software are temporal requirements. To specify temporal requirements for process-oriented programs, an approach in which requirements are specified as control loop invariants is proposed in [2]. When describing requirements a program is considered as a black box, i. e. the requirements do not contain information about program structure (process states, values of process timers and local variables). However, such information is needed for proving verification conditions. Therefore, we present a control loop invariant in the form of conjunction of a formalized requirement and an extra invariant containing information about the program structure. We specify requirements and extra invariants in the previously developed temporal requirement language DV-TRL [8] that is a variant of typed first-order logic. This language is based on the *update state* data type values of which represent histories of all changes in a program. Specialized functions allow using variables values at different points in time in requirements. It gives the opportunity to specify temporal requirements.

Only verification condition generation can be fully automated. The problems of loop invariant synthesis and proving verification conditions are undecidable in general. Nevertheless there are approaches to solving these problems in particular cases.

Earlier we developed a set of temporal requirement patterns in our language DV-TRL [7]. For each requirement pattern, a corresponding extra invariant pattern used for specifying requirement-dependent invariants and a set of lemmas needed for proving verification conditions were defined. We also developed a set of requirement-independent extra invariant patterns. This allows automating deductive verification of process-oriented programs with requirements satisfying these requirement patterns. However there are requirements that do not satisfy previously developed patterns. It has been noted that previously developed patterns and patterns that could describe new classes of requirements can be made up of a small number of basic patterns. This paper presents an approach to automation of deductive verification of process-oriented programs in which requirement patterns can be constructed by combining basic patterns and corresponding extra invariant patterns and lemmas with their proofs can be generated automatically.

This paper has the following structure. Section 2 describes our approach to automation of deductive verification. Section 3 demonstrates our approach on an example. Section 4 discusses related works on automated loop invariant generation. Section 5 summarizes the results.

2. Approach to Automation of Deductive Verification

This section describes our approach to automation of deductive verification of process-oriented programs based on patterns and lemmas. The relationship between different kinds of patterns and lemmas is shown in Figure 1. In this approach, patterns are used to represent requirements and extra invariants. We use requirement patterns to specify requirements and extra invariant patterns to specify extra invariants. Extra invariants and their patterns are divided into requirement-dependent and requirement independent ones. For each program, several requirement-independent extra invariants can be defined. For each requirement, one requirement-dependent extra invariant is defined. Requirement patterns and requirement-dependent extra invariant patterns are divided into basic and derived ones. All derived requirement patterns and extra invariant patterns are defined by combining basic requirement and extra invariant patterns respectively. Each basic (derived) requirement pattern has a corresponding basic (derived, respectively) extra invariant pattern and a set of lemmas associated with it. Lemmas for derived patterns are proved using lemmas for basic patterns and used for proving verification conditions. This allows automatically constructing the derived extra invari-

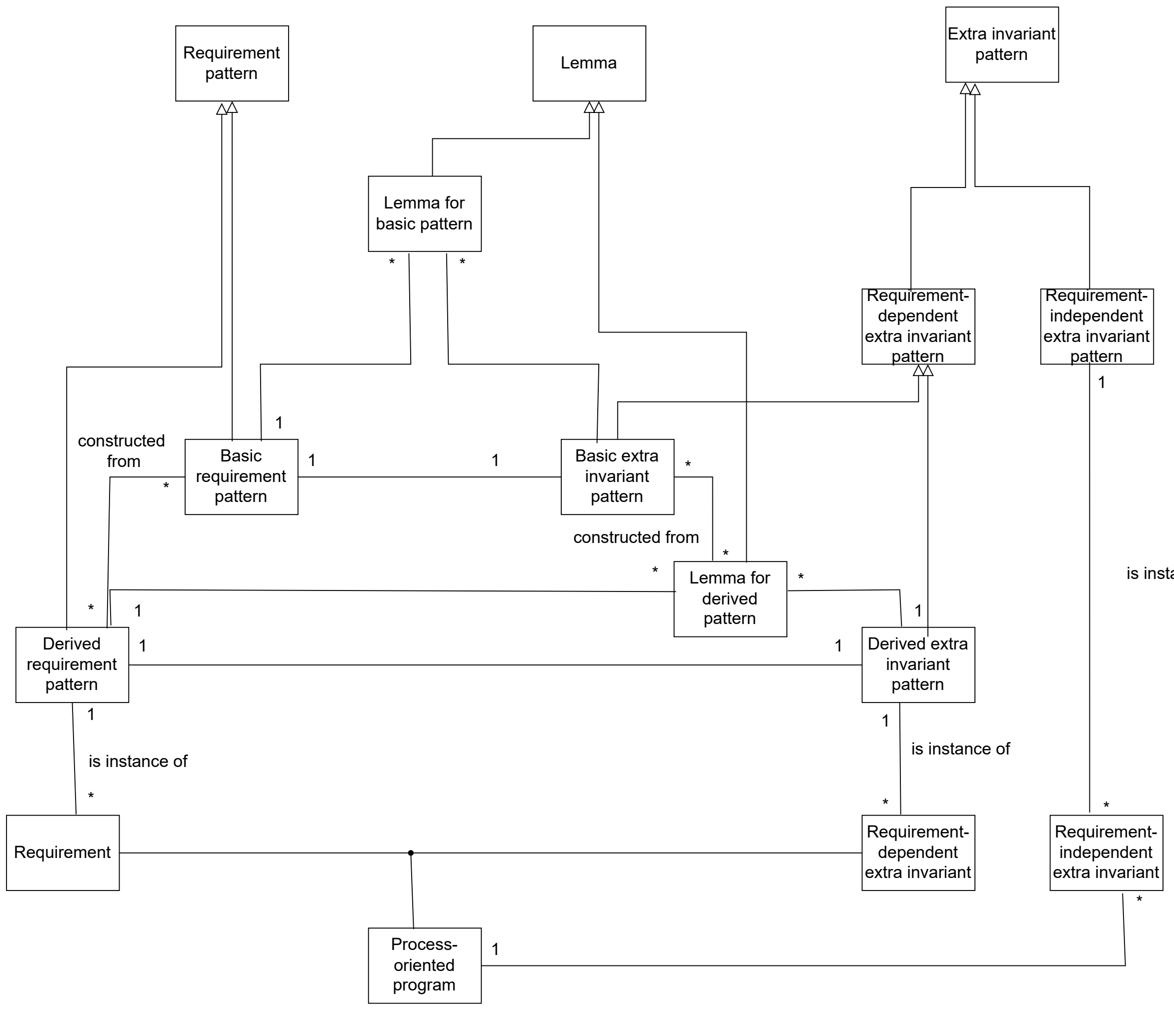
Requirement pattern
Lemma
Extra invariant pattern
Lemma for basic pattern
Requirement-dependent extra invariant pattern
Requirement-independent extra invariant pattern
constructed from
Basic requirement pattern
Basic extra invariant pattern
constructed from
Lemma for derived pattern
is instance of
Derived requirement pattern
Derived extra invariant pattern
is instance of
is instance of
Requirement
Requirement-dependent extra invariant
Requirement-independent extra invariant
Process-oriented program
1
*

Fig. 1. Relationship between different kinds of patterns and lemmas.

ant pattern corresponding to the derived requirement pattern and lemmas using the definition of the requirement pattern as well as automatically proving these lemmas. Thus, an extra invariant is a conjunction of requirement independent invariants and a requirement dependent invariant. For proving verification conditions, we define proof scripts. These scripts, together with the lemmas allow automatically proving verification conditions.

Earlier we developed a verification condition generator for programs in poST language [28]. In this work, we have developed algorithms for constructing extra invariant patterns for derived requirement patterns, generating lemmas for them and proving these lemmas. We plan to develop a verification tool based on these algorithms and the verification condition generator.

User interaction with the verification tool is shown in Figure 2. First, the user determines requirement-independent extra invariants based on the process-oriented program being verified. To specify each invariant, the user selects a requirement-independent extra invariant pattern and specifies parameter values for it. Then the following actions are performed for each requirement to be verified. The user selects a derived requirement pattern or creates it using basic patterns if there is no appropriate pattern. In the latter case, the verification tool generates the corresponding derived extra invariant pattern and lemmas that are proved in Isabelle/HOL [20]. For each derived requirement pattern, there is a natural language description of behavior that can be specified using this pattern and examples of known uses. For each basic requirement pattern, there is a natural language description of propositions specified by this pattern and examples of derived patterns in the definitions of which this basic pattern is used. This information allows a user to identify similar requirements and choose a pattern.

Each basic pattern is parameterized by two update states: s_1 (an update state in which the pattern instance should be true) and s (the update state in which the loop invariant should be true). The definition of a basic requirements pattern R has the following form:

$$R \equiv \lambda s.\lambda(s_1, p_1, ..., p_m, A_1, ..., A_n).$$
$$R'(s_1, s, p_1, ..., p_m, A_1(s, r_{j_1}), ..., A_n(s, r_{j_n})),$$

where p_1,..., p_m are constant parameters, R' is a parameterized formula of the DV-TRL language without negations in which the formula parameters A_1,..., A_n do not appear in premises of implications, r_{j_i} (i=1,..., n) are

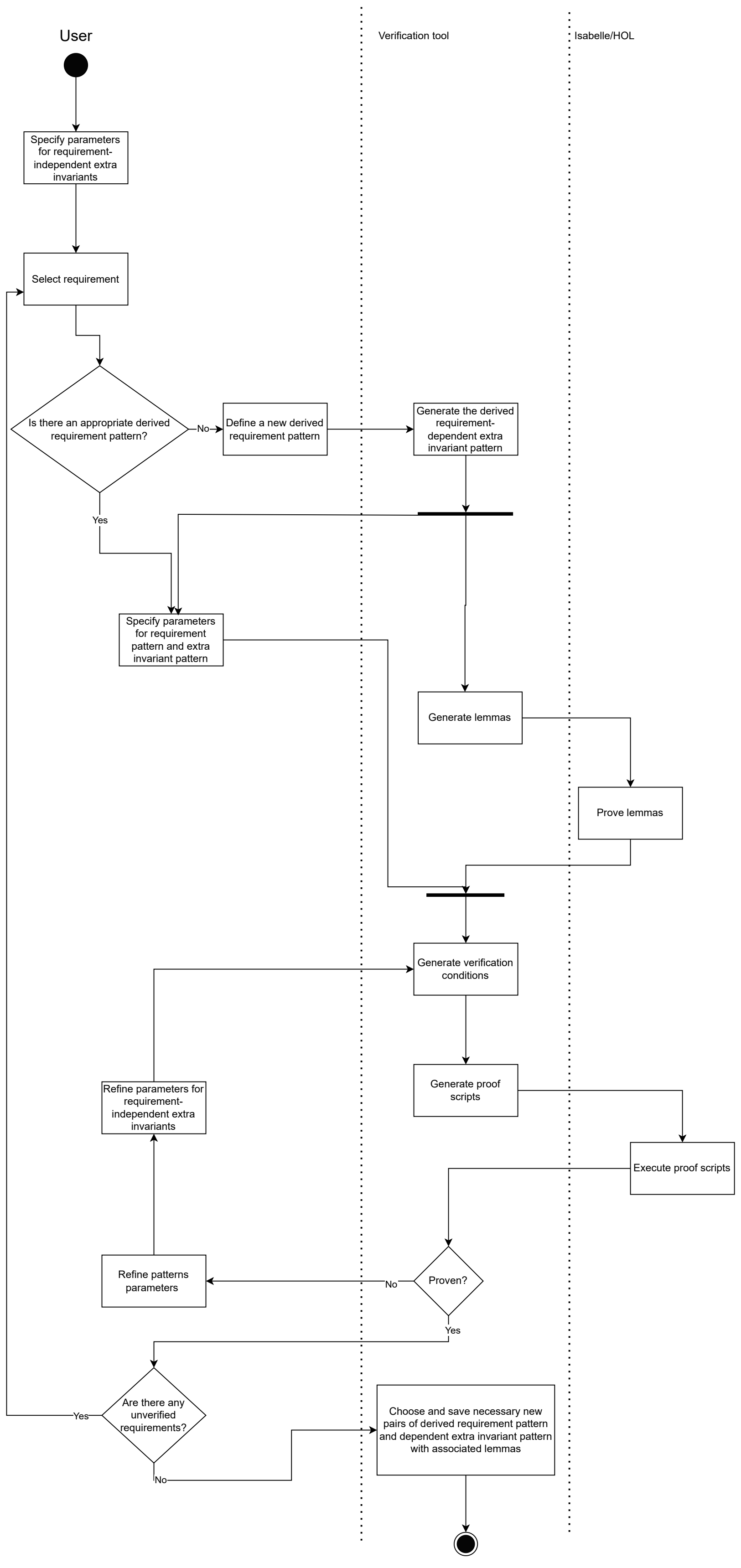
User
Verification tool
Isabelle/HOL
Specify parameters for requirement-independent extra invariants
Select requirement
Is there an appropriate derived requirement pattern?
No
Yes
Define a new derived requirement pattern
Generate the derived requirement-dependent extra invariant pattern
Specify parameters for requirement pattern and extra invariant pattern
Generate lemmas
Prove lemmas
Generate verification conditions
Generate proof scripts
Refine parameters for requirement-independent extra invariants
Execute proof scripts
Proven?
No
Yes
Refine patterns parameters
Are there any unverified requirements?
Yes
No
Choose and save necessary new pairs of derived requirement pattern and dependent extra invariant pattern with associated lemmas

Fig. 2. User interaction with the verification tool.

variables of the update state data type bound
by quantifiers in R' such that $r_{j_i} \leq s$ and A_1,..., A_n are formula parameters whose values are requirement parameter formulas defined as follows: 1) atomic formulas and their negations are requirement parameter formulas, 2) if A_1 and A_2 are requirement parameter formulas, then $\lambda(s, s_1).A_1(s, s_1) \wedge A_2(s, s_1)$ and $\lambda(s, s_1).A_1(s, s_1) \vee A_2(s, s_1)$ are requirement parameter formulas, 3) if P is a requirement pattern with m' constant parameters and n' formula parameters, p_1,..., $p_{m'}$ are constants of the appropriate types and A_1,..., $A_{n'}$ are requirement parameter formulas, then $\lambda(s, s_1).P(s, s_1, p_1, ..., p_{m'}, A_1, ..., A_{n'})$ is a requirement parameter formula.

The definition of a derived pattern has the following form:

$$R \equiv \lambda s.\lambda(p_1, ..., p_m, A_1, ..., A_n).R'(s, p_1, ..., p_m, A_{h_1}, ..., A_{h_u}, A_{j_1}(s), ..., A_{j_v}(s)),$$

where s is the update state in which the requirement should be fulfilled; p_1,..., p_m are constant parameters; A_{j_1},.., A_{j_v} are formula parameters that are requirement parameter formulas; A_{h_1},..., A_{h_u} are formula parameters whose values are pattern parameter formulas that are defined as follows: 1) atomic formulas parameterized by one update state and their negations are pattern parameter formulas, 2) if A_1 and A_2 are pattern parameter formulas then $\lambda s_1.A_1(s_1) \wedge A_2(s_1)$ and $\lambda s_1.A_1(s_1) \vee A_2(s_1)$ are pattern parameter formulas; R' is a parameterized formula that has the form $P(s, p_1, ..., p_{m'}, A_1, ..., A_{n'})$ where P is a derived requirement pattern with m' constant parameters and n' formula parameters, p_1,..., $p_{m'}$ are constants of the appropriate types and A_1,..., $A_{n'}$ are parameterized formulas of the appropriate types. A derived requirement pattern can also be combined with other patterns. But in general, the value of not every parameter of a derived pattern may contain nested patterns. In this scheme, only the values of the the parameters A_{j_1},..., A_{j_r} ($\{j_1; ...; j_r\} \subset \{1; ...; n\}$ can contain nested patterns. Values of other formula parameters A_{h_1},..., A_{h_u} ($r + u = n$) cannot contain nested patterns.

Next, the user specifies parameter values for these patterns. After the requirement has been formalized and a requirement-dependent extra invariant has been defined using patterns, the verification tool generates verification conditions based on the process-oriented program and proof scripts for proving the verification conditions. These proof scripts are executed in the Isabelle/HOL. If the verification conditions have been proved, the user proceeds to verification of another requirement, if any. If some verification conditions are not proved, the user refines pattern parameters, and the verification tool proceeds to re-generation of verification conditions. If there are no unverified requirements, the verification tool saves necessary pairs of derived requirement and extra invariant patterns with associated lemmas, and the verification of the program is completed.

We refer a reader to our report[1] for more details about our approach. In this paper, we only demonstrate our approach on an example.

3. Example

In this section, we consider an example of constructing a derived requirement and the corresponding extra invariant pattern and specifying a requirement and extra invariants according the patterns.

Let us consider a hand dryer control program as an example. The hand dryer includes a sensor indicating whether there are hands, a fan and a heater. The program receives the input signal from the sensor and, depending on the input signal, controls the fan heater. If the hands appear, the fan heater turns on. If the hands are removed, then after a certain time the fan heater turns off.

The hand dryer control program is presented on Figure 3:

```
PROGRAM Controller
  VAR_INPUT
    hands : BOOL;
  END_VAR
  VAR_OUTPUT
    dryer : BOOL;
  END_VAR
  PROCESS Ctrl
    STATE waiting
      IF hands  THEN
        dryer := TRUE;
        SET NEXT;
      ELSE
        dryer := FALSE;
      END_IF
    END_STATE
    STATE drying
      IF hands  THEN
        RESET TIMER;
      END_IF
      TIMEOUT T#1s THEN
        SET STATE waiting;
      END_TIMEOUT
    END_STATE
  END_PROCESS
END_PROGRAM
```

Fig. 3. Hand dryer control program in poST language.

Two variables are declared in the program: the input variable `hands`, which shows the presence of hands under the fan heater, and the output variable `dryer`, which determines whether the fan heater is turned on. One process `Ctrl` is defined. It has two states `waiting` and `drying`. In the state `waiting`, the presence of hands is checked. If there are hands, the fan heater turns on and the process `Ctrl` transits to the state `drying`. If the hands are absent, the fan heater turns off. In the state `drying`, the presence of hands is also checked. If there are hands, the timer of the process is reset. The timeout is set in this state. After 1 second, the process `Ctrl` transits to the state `waiting`.

The following requirement on the hand dryer control program is needed to be verified: "If there are no hands, then the fan heater should turn off after no more than 1 second if the hands do not reappear during this time".

The following designations are used in the patterns discussed below:

- s, $s_1 \ldots s_n$, r, $r_1 \ldots r_m$ are states;
- A_1,A_2,A_3 are arbitrary logical formulas;

[1]https://github.com/ivchernenko/PSSV2024-report

- $p(s)$ returns the previous *external* state. An *external* state is a state at the point of transfer of variable values from controller to control object in the control loop. Otherwise, the state is *internal*;
- $s \leq r$ returns true if $s = r$, or $s \leq p(r)$;
- $s_1 \leq \ldots \leq s_n$ is short for $s_1 \leq s_2 \wedge \ldots \wedge s_{n-1} \leq s_n$;
- $s < r$ returns true if $s \leq r$ and $s \neq r$
- $e(s)$ returns true if state s is external;
- $n(s_1, s_2)$ returns the number of external states between states s_1 and s_2;
- $s[x]$ is a value of program variable x in state s;
- $getPstate(s, p)$ is the state of process p in update state s;
- $ltime(s, p)$ is the value of the timer of process p in state s;
- $consecutive(s_1, s_2)$ returns true if $e(s_1) \wedge e(s_2) \wedge s_1 \leq s_2 \wedge n(s_1, s_2) = 1$.

Let us describe verification of the program step by step.

The first step of verification is determining requirement independent extra invariants. One of the requirement-independent extra invariant patterns states that if process p is in state q, variable x has value v. This pattern is defined as follows:

$$\lambda s.getPstate(s, p) = q \longrightarrow s[x] = v$$

The following property of the hand dryer control program can be specified using this pattern: "If the process `Ctrl` is in the state `drying`, the variable `dryer` has the value `TRUE`". The parameters have the following values:

```
p ≡ Ctrl; q ≡ drying; x ≡ dryer; v ≡ True.
```

Next, the verification of the requirement formulated above is performed. This requirement satisfies the following derived requirement pattern: "If event A_1 has occurred, then event A_3 should occur no more than after time t and after the occurrence of A_1 and before the occurrence of A_3, the condition A_2 should be true". This pattern is defined as follows:

$$DRP(s, t, A_1, A_2, A_3) \equiv$$
$$BRP2(s, s, (\lambda r_2 r_1.\neg A_1(r_1) \vee BRP1(r_2, r_1, t, A_2, A_3))).$$

Here, s is an update state in which the requirement should be satisfied, t is a constant parameter, A_1, A_2 and A_3 are formula parameters, and a value of A_1 cannot contains nested patterns and values of A_2 and A_3 can. In this definition, the following two basic requirement patterns `BRP1` and `BRP2` defined in the knowledge base are used.

The first pattern `BRP1` defined below asserts that no later than time t after the start of the time counting in state s_1, event A_2 will occur and from the start of the time counting to the occurrence of event A_2, condition A_1 is fulfilled.

$$
\begin{aligned}
&BRP1(s, s_1, t, A_1, A_2) \equiv\\
&n(s_1, s) \geq t \longrightarrow\\
&(\exists r_2.e(r_2) \wedge s_1 \leq r_2 \leq s \wedge n(s_1, r_2) \leq t \wedge A_2(s, r_2) \wedge\\
&\quad (\forall r_1.(e(r_1) \wedge s_1 \leq r_1 \leq r_2 \wedge r_1 \neq r_2 \longrightarrow A_1(s, r_1)))),
\end{aligned}
$$

In this definition, s and s_1 are the update states in which a control loop invariant and the pattern instance are satisfied respectively, t is a constant parameter, and A_1 and A_2 are formula parameters. The inequality $n(s_1, s) \geq t$ in the premise is necessary because if time t has not passed since the start of the countdown in state s_1, then event A_2 may not occur until the final state s. This definition asserts that there is an external state r_2 between the states s_1 and s such that the time elapsed from state s_1 to state r_2, no more than t and A_2 is satisfied in the state r_2. Moreover, for each external state r_1 between the states s_1 and r_2, including s_1, but excluding r_2, the condition A_1 is satisfied.

The second pattern BRP2 defined below asserts that a condition A_1 should always be true between iterations of the control loop up to the current state s_1.

$$
\begin{aligned}
&BRP2(s, s_1, A_1) \equiv\\
&\forall r_1.e(r_1) \wedge r_1 \leq s_1 \longrightarrow A_1(s, r_1)
\end{aligned}
$$

In this definition, s and s_1 are the update states in which a control loop invariant and the pattern instance are satisfied respectively, A_1 is a formula parameter. This definition asserts that the condition A_1 is satisfied in each external update state up to the state s_1.

In the definition of the derived requirement pattern DRP, the instance of the pattern BRP2 is satisfied in the state s because the instance is the control loop invariant. The bound variables r_2 and r_1 correspond to the update states in which the control loop invariant and the parameter A_1 of the pattern BRP2 are satisfied respectively. The value of the parameter A_1 of the basic pattern BRP2 is the disjunction. Its first disjunct is the negation of the derived requirement pattern parameter A_1. The second disjunct is an instance of the pattern BRP1 that is satisfied in the update state r_1 and the values of the parameters t, A_1 and A_2 of which are the parameters t, A_2 and A_3 of the derived pattern respectively.

The basic extra invariant pattern BIP1 corresponding to the basic requirement pattern BRP1 is defined as follows:

$$
\begin{aligned}
&BIP1(s, s_1, t, t_1, A_1, A_2)\\
&(\exists r_2.e(r_2) \wedge s_1 \leq r_2 \wedge r_2 \leq s \wedge n(s_1, r_2) \leq t \wedge A_2(s, r_2) \wedge\\
&\quad (\forall r_1.e(r_1) \wedge s_1 \leq r_1 \wedge r_1 < r_2 \longrightarrow A_1(s, r_1))) \vee
\end{aligned}
$$

$n(s_1, s) < t_1(s) \wedge$
$(\forall r_1 . e(r_1) \wedge s_1 \leq r_1 \wedge r_1 \leq s \longrightarrow A_1(s, r_1))$

An instance of this extra invariant pattern asserts that the corresponding instance of the requirement pattern is fulfilled, but it also additionally asserts that the maximum waiting time for the event A_2 in state s is $t_1(s)$. In this definition, s and s_1 are the update states in which a control loop invariant and the pattern instance are satisfied respectively, t is a constant parameter, its value in an instance of the pattern BIP1 is equal to the value of the parameter t in the corresponding instance of the pattern BRP1, t_1 is an additional parameter that is a function depending on s, A_1 and A_2 are formula parameters, their values are not equal in general, but related to the values of the parameters A_1 and A_2 respectively in the corresponding instance of the pattern BRP1. This definition asserts that either there is an external state r_2 between the states s_1 and s such that the time elapsed from state s_1 to state r_2 is no more than t, A_2 is satisfied in the state r_2 and for each external state r_1 between the states s_1 and r_2, including s_1, but excluding r_2, the condition A_1 is satisfied or the time $t_1(s)$ has not passed after the start of the countdown in the state s_1 and for each external state r_1 between s_1 and s, the condition A_1 is satisfied in r_1.

The basic extra invariant pattern BIP2 corresponding to the basic requirement pattern BRP2 coincides with BRP2 for $s_1 = s$, i. e., $BIP2$ is defined as follows:

$BIP2(s, A'_1) \equiv$
$\forall r_1 . e(r_1) \wedge r_1 \leq s \longrightarrow A'_1(s, r_1)$

This pattern is parameterized by one update state s because an instance of this pattern is an extra invariant, and it is satisfied in the state s. The value of the parameter A_1 in the pattern BIP2 is not equal in general, but related to the value of the parameter A_1 in the corresponding instance of of the pattern BRP2.

After the user has defined the derived requirement pattern, the corresponding derived extra invariant pattern DIP is constructed. It is defined as follows:

$DIP(s, t, t_1, A_1, A_2, A_3) \equiv$
$BIP2(s, (\lambda r_2 r_1 . \neg A_1(s_1) \vee BIP1(r_2, r_1, t, t_1, A_2, A_3)))$

In this definition, s is the update state in which the extra invariant is satisfied, t is a constant parameter, its value is equal to the value of the parameter t in the corresponding instance of the pattern DRP, t_1 is an additional parameter that is a function depending on the update state s, A_1, A_2 and A_3 are formula parameters, and the value of the parameter A_1 is equal to the value of the parameter A_1 in the corresponding instance of the pattern DRP and the values of A_2 and

A_3 are not equal in general, but related to the values of the parameters A_2 A_3 respectively in the corresponding instance of the pattern DRP. The instance of the pattern BIP2 is satisfied in the state s. The bound variables r_2 and r_1 correspond to the update states in which the control loop invariant and the parameter A_1 of the pattern BIP2 are satisfied respectively. The value of the parameter A_1 in the pattern BIP2 is the disjunction. The first disjunct is the negation of the deriverived pattern parameter A_1. The second disjunct is the instance of the pattern BIP1 that is satisfied in the state r_1. The values of the parameters t, t_1, A_1 and A_2 in the pattern BIP1 are parameters t, t_1, A_2 and A_3 in the derived pattern DIP.

After defining the derived requirement and extra invariant patterns, the user specifies the following pattern parameters:

$A_1 \equiv \lambda(s, r_1).r_1[hands] = False;$

$A_2 \equiv \lambda(s, r_2).r_2[dryer] = True \wedge r_2[hands] = False;$

$A_3 \equiv \lambda(s, r_3).r_3[dryer] = False \vee r_3[hands] = True;$

$t \equiv 10;$

$t_1 \equiv \lambda s.$ if $getPstate(s, Ctrl) = drying$ then $ltime(s, Ctrl)$ else 10.

Next, lemmas for these derived patterns are generated and proved in Isabelle/HOL, verification conditions and proof scripts for them are generated. These proof scripts are executed in Isabelle/HOL, and all verification conditions are proved. Since this derived requirement pattern is common, it is saved to the knowledge base along with the associated derived extra invariant pattern and the lemmas.

4. Related Work

There is a wide variety of methods of finding loop invariants. These include abstract interpretation [10], induction-iteration method [26], template-based methods [9], recurrence analysis [15], using failed proof attempts [25] and invariant strengthening on demand [16], dynamic analysis [21] and machine learning [22]. Abstract interpretation and template-based methods are the most common approaches to the static loop invariant inference [12]. Let us consider the works closest to this one on the automatic generation of loop invariants.

Template-based methods of loop invariant generation are most successfully applied within the domain of linear arithmetic [6]. In [9], a method of linear loop invariants generation based on templates is proposed. Invariants have the form of linear inequalities. The authors generate constraints on the template parameters that are coefficients in the inequality. These constraints ensure that the invariant is true when the program enters the loop and after an iteration if it was true before the iteration. Farkas' Lemma is used to generate the constraints. The obtained constraints can be solved by quantifier elimination. But because quantifier elimination is a costly process, the authors simplify the constraints using various techniques. In our work, extra invariants are not linear inequalities relating the values of program variables in one point, but formulas relating the values of the variables at different points in time and containing quantifiers over update states (in patterns). Currently the values of pattern parameters are specified manually, but our lemmas can be used to generate constraints on our pattern parameters. In this case, the constraints will be quantifier-free. We plan to investigate the problem of generating the constraints in the future.

To find the values of template parameters, SMT solvers can be used. In [24], an approach based on user-defined templates is proposed. The authors reduce the problem of searching the template parameters values to satisfiability solving, that allow using off-the-shelf solvers. The values of template parameters are not constants, but expressions and predicates. To find such values, the authors make assumptions about the domain that allow them to reduce this problem to finding constants. An SMT solver is used to obtain these values. In our work, the values of the pattern parameters are conditional expressions including not only constants, but also terms containing process timers. To reduce the problem of finding such expressions to finding constants, we could consider instances of constraints for different paths in the program.

In STeP [17], two approaches to invariant generation are used: the bottom-up approach in which invariants are generated by static analysis of the program and the top-down approach that is goal-oriented. In the top-down approach, unproven verification conditions are used to strengthen invariants. If some verification condition cannot be proven, the weakest precondition with respect to the invariant to be proven and the transition that the verification condition corresponds to is computed. The strengthened invariant is the conjunction of the original invariant to be proven and this weakest precondition. In our work, we also use both bottom-up and top-down (i. e., requirement-dependent) invariants. We could also use invariant strengthening. This approach would need to be used together with the heuristic of replacing a constant with a term. But it was noted that extra invariants needed for proving requirements satisfying the same pattern are similar and can also be described by a pattern.

This study [3] is devoted to the deductive verification of programs written in the LD language from the IEC 61131-3 standard. Temporal requirements for LD programs are specified using timing charts. The verification process employs the Why3 deductive verification system. The authors formalized LD instructions as functions within Why3. In this framework, an event and the subsequent stable state in a timing chart are modeled as a loop in Why3, where the loop's body corresponds to one iteration of the LD program's control loop, and the loop guard represents the condition that the input must meet at the moment of the event and during the stable state. The verified requirements are framed as invariants of this loop. To model fixed-duration sequences of events, a time counter is introduced, which increments with each iteration. If certain verification conditions cannot be established, counterexamples are generated for further analysis. Since the invariants are insufficient, the authors use automatic generation of additional loop invariants. The authors use the abstract interpretation method to automatically generate loop invariants. A former prototype for Why3 system does not support boolean variables that appear in programs representing LD programs and timing charts in Why3. The authors encode Boolean variables as integer variables with constraints that allow using existing methods to generate loop invariant with Boolean variables. In our work, we verify programs in more expressive process-oriented languages. To specify requirements, we use first-order logic instead of timing charts. We represent a program as one infinite control loop, not as several loops. We have also noted that extra invariant patterns can be defined to specify auxiliary properties and use these patterns instead of the abstract interpretation.

The paper [5] explores the auto-active method for automating deductive verification. This approach requires users to supply supplementary guiding annotations, such as assertions, ghost code, and lemma functions, to achieve a higher degree of proof automation. As a result, it enables the use of automatic solvers in scenarios where interactive provers employed traditionally. The authors implement auto-active verification for C programs within the Frama-C framework. In our study, we do not use ghost code and lemma functions. However, we can incorporate formalized requirements as annotations in the program. For instance, the control loop invariant INV at the start of an iteration can be expressed with the annotation `ASSUME INV`, while the invariant at the end can be represented with `ASSERT INV`. Additionally, we can use the annotation `ASSERT` to add extra assertions at any point in the program.

In [19], an approach that allows one to make requirement specifications reusable using object-oriented concepts. In this approach, in addition to declarative specifications, a subset of the programming language is used to set requirements. To specify temporal requirements, loops with loop invariants and variants are used. The authors chose Eiffel as a programming language. Their approach is based on the specification drivers that are routines provided with contracts and capture some behavioral properties of their formal parameters through the contracts.

Then the authors describe requirement patterns using classes called seamless object-oriented requirement templates. Each such class contains a specification driver and deferred features corresponding to the requirement pattern parameters. To create a requirement from a pattern, a class presenting the requirement and called "seamless object-oriented requirement" inheriting from the class representing the requirement pattern and implementing the deferred features is created. In our work, we verify programs in process-oriented languages, not object-oriented languages. We also define requirement patterns, but define their in the typed first-order logic and do not use a programming language in requirements. We specify temporal requirements as invariants of the control loop that is a concept in control software.

In [13], a template-based method is combined with abstract interpretation and dynamic analysis to generate loop invariants. A template is a Boolean combination of linear inequalities. Each path in the program satisfies the inductiveness condition: if the corresponding template instance is true at the beginning of the path, the corresponding template instance must be true at the end of the path. This inductiveness condition is translated into a constraint. Constraints are non-linear and difficult to solve. Therefore, static and dynamic analysis is used. Test executions for the dynamic analysis are generated by other tools. Concrete or symbolic execution can be used in the dynamic analysis. In the dynamic analysis, program variables in templates are replaced with their values. This allows one to obtain linear constraints. First, abstract interpretation is applied to generate some invariant that are not sufficient, and then other invariants are generated in a goal directed way. In our work, we use only pattern-based method without combining it with other methods. Similar to that work, we use both requirement-independent invariants that are not goal directed and the requirement-dependent invariants that are goal directed.

The paper [4] presents a template-bas4ed approach to loop invariant generation in the combined theory of linear arithmetic and uninterpreted function symbols. First, the authors apply purification, i. e., replacing subterms that are application of an uninterpreted function to expressions with a new variable with saving the definition of this variable. Then constraints are generated and solved. Our invariants are in the theory of update states possibly combined with the theory of arithmetic. Using our lemmas and some heuristics that we plan to develop, we could generate constraints that do not contain update states.

In [23], a method of loop invariant generation combining template-based approach and predicate abstraction is proposed. The authors developed three algorithms: two algorithms iteratively compute fixed-points, and one algorithm uses constraint solving. An invariant solution is a mapping each unknown in templates to some set of predicates such that the verification conditions are true. The authors use SMT solvers to find invariant solutions.

5. Conclusion

In this paper, we have presented an approach to deductive verification of process-oriented programs in which temporal requirements are specified using combination of basic patterns. In this approach, a set of basic requirement patterns is defined. For each such basic pattern, the corresponding basic extra invariant pattern and lemmas are defined. Then the basic requirement patterns can be combined to define derived requirement patterns. For each derived requirement pattern, the corresponding extra invariant pattern and lemmas for proving verification conditions are constructed.

The approach proposed in this paper will allow one to automatically determine the extra invariant pattern and lemmas needed to prove verification conditions for a given requirement and prove these lemmas. Having generalized previously developed strategies for proving verification conditions so that the strategies are parameterized by the appropriate lemma, we can automate proving verification conditions. Thus the only task that has not yet been automated is finding values of parameters of an extra invariant pattern.

Currently, our basic requirement pattern set contains 9 patterns. Using them, we have defined 11 common derived patterns defining classes including at least two requirements and 4 special derived patterns. This patterns have allowed us to specify and verify all requirements from our collection containing 76 requirements. However, our pattern system currently does not allows one to specify some classes of requirements, for example, requirements that state that some event should or should not happen within a time interval after or before some other event considered in [18] as well as requirements stating that some event should not happen after (before) some delay after (before) some other event. Also requirements stating that an event must occur k times considered in [11] cannot be easily specified. We plan extend our pattern system in the future to cover these classes of requirements.

In the future, we also plan to develop tools for generation of the derived extra invariant patterns, the lemmas and scripts for proving these lemmas as well as scripts for proving verification conditions. We also plan to develop a heuristic algorithm for finding the value of the parameters of extra invariant patterns.

References

1. Programmable Controllers—Part 3: Programming Languages, document IEC 61131-3, International Electrotechnical Commission, 2013.
2. Anureev I., Garanina N., Liakh T. et al. Two-step deductive verification of control software using Reflex //Perspectives of System Informatics: 12th International Andrei P. Ershov Informatics Conference, PSI 2019, Novosibirsk, Russia, July 2—5, 2019, Revised Selected Papers 12. — Springer International Publishing, 2019. — P. 50-63. https://doi.org/10.1007/978-3-030-37487-7_5
3. Belo Lourenço C., Cousineau D., Faissole F. et al. Automated formal analysis of temporal properties of Ladder programs //International Journal on Software Tools for Technology Transfer. — 2022. — Vol. 24. — №. 6. — P. 977-997. https://doi.org/10.1007/s10009-022-00680-0
4. Beyer D., Henzinger T. A. Majumdar R., Rybalchenko A. Invariant synthesis for combined theories //International Workshop on Verification, Model Checking, and Abstract Interpretation. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2007. — P. 378-394. https://doi.org/10.1007/978-3-540-69738-1_27
5. Blanchard A., Loulergue F., Kosmatov N. Towards full proof automation in Frama-C using auto-active verification //NASA Formal Methods Symposium. — Cham : Springer International Publishing, 2019. — P. 88-105. https://doi.org/10.1007/978-3-030-20652-9_6
6. Breck J., Cyphert J., Kincaid Z., Reps T. Templates and recurrences: better together //Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. — 2020. — P. 688-702. https://doi.org/10.1145/3395656
7. Chernenko I. M. Requirements patterns in deductive verification of process-oriented programs and examples of their use //System Informatics. — 2023. — №. 22. — P. 11-20. https://doi.org/10.31144/si.2307-6410.2023.n22.p11-20
8. Chernenko I., 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). — IEEE, 2022. — P. 657-662. https://doi.org/10.1109/EDM55285.2022.9855145
9. Colón M. A., Sankaranarayanan S., Sipma H. B. Linear invariant generation using non-linear constraint solving //Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15. — Springer Berlin Heidelberg, 2003. — P. 420-432. https://doi.org/10.1007/978-3-540-45069-6_39
10. Cousot P., Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints //Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. — 1977. — P. 238-252. https://doi.org/10.1145/512950.512973
11. Dwyer M. B., Avrunin G. S., Corbett J. C. Patterns in property specifications for finite-state verification //Proceedings of the 21st international conference on Software engineering. — 1999. — P. 411-420. https://doi.org/10.1145/302405.302672

12. Furia C. A., Meyer B., Velder S. Loop invariants: Analysis, classification, and examples //ACM Computing Surveys (CSUR). — 2014. — Vol. 46. — №. 3. — P. 1-51. https://doi.org/10.1145/2506375
13. Gupta A., Rybalchenko A. Invgen: An efficient invariant generator //Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26-July 2, 2009. Proceedings 21. — Springer Berlin Heidelberg, 2009. — P. 634-640. https://doi.org/10.1007/978-3-642-02658-4_48
14. Hähnle R., Huisman M. Deductive software verification: from pen-and-paper proofs to industrial tools //Computing and Software Science: State of the Art and Perspectives. — 2019. — P. 345-373. https://doi.org/10.1007/978-3-319-91908-9_18
15. Kovács L. Reasoning algebraically about P-solvable loops //International Conference on Tools and Algorithms for the Construction and Analysis of Systems. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. — P. 249-264. https://doi.org/10.1007/978-3-540-78800-3_18
16. Leino K. R. M., Logozzo F. Loop invariants on demand //Asian symposium on programming languages and systems. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2005. — P. 119-134. https://doi.org/10.1007/11575467_9
17. Manna Z., Bjørner N., Browne A. et al. STeP: The stanford temporal prover //TAPSOFT'95: Theory and Practice of Software Development: 6th International Joint Conference CAAP/FASE Aarhus, Denmark, May 22—26, 1995 Proceedings 20. — Springer Berlin Heidelberg, 1995. — P. 793-794. https://doi.org/10.1007/3-540-59293-8_237
18. Mekki A., Ghazel M., Toguyeni A. Patterns-Based Assistance for Temporal Requirement Specification //Proceedings of the International Conference on Software Engineering Research and Practice (SERP). — The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2011. — P. 1. https://www.researchgate.net/profile/Armand-Toguyeni/publication/260420194_Patterns-Based_Assistance_for_Temporal_Requirement_Specification/links/549a90c30cf2d6581ab16f9c/Patterns-Based-Assistance-for-Temporal-Requirement-Specification.pdf (online; accessed: 02.10.2024)
19. Naumchev A. Seamless object-oriented requirements //2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON). — IEEE, 2019. — P. 0743-0748. https://doi.org/10.1109/SIBIRCON48586.2019.8958211
20. Paulson L. C., Nipkow T., Wenzel M. From LCF to isabelle/hol //Formal Aspects of Computing. — 2019. — Vol. 31. — P. 675-698. https://doi.org/10.1007/s00165-019-00492-1
21. Perkins J. H., Ernst M. D. Efficient incremental algorithms for dynamic detection of likely invariants //proceedings of the 12th ACM SIGSOFT twelfth International Symposium on Foundations of Software Engineering. — 2004. — P. 23-32. https://doi.org/10.1145/1029894.1029901
22. Si X., Dai H., Raghothaman M. et al. Learning loop invariants for program verification //Advances in Neural Information Processing Systems. — 2018. — Vol. 31. https://proceedings.neurips.cc/paper_files/paper/2018/file/65b1e92c585fd4c2159d5f33b5030ff2-Paper.pdf (online; accessed: 02.10.2024)
23. Srivastava S., Gulwani S. Program verification using templates over predicate abstraction //Pro-

ceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. — 2009. — P. 223-234. https://doi.org/10.1145/1542476.1542501

24. Srivastava S., Gulwani S., Foster J. S. Template-based program verification and program synthesis //International Journal on Software Tools for Technology Transfer. — 2013. — Vol. 15. — P. 497-518. https://doi.org/10.1007/s10009-012-0223-4
25. Stark J., Ireland A. Invariant discovery via failed proof attempts //International Workshop on Logic Programming Synthesis and Transformation. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1998. — P. 271-288. https://doi.org/10.1007/3-540-48958-4_15
26. Suzuki N., Ishihata K. Implementation of an array bound checker //Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. — 1977. — P. 132-143. https://doi.org/10.1145/512950.512963
27. Zyubin V. E. Hyper-automaton: A model of control algorithms //2007 Siberian Conference on Control and Communications. — IEEE, 2007. — P. 51-57. https://doi.org/10.1109/SIBCON.2007.371297
28. 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. – T. 10. – C. 35238-35250. https://doi.org/10.1109/ACCESS.2022.3157601