Extra Invariant Patterns and Theory for Temporal Requirements in Deductive Verification of Process-Oriented Programs*

Ivan M. Chernenko[1][0000−0001−7675−8449]

Institute of Automation and Electrometry, Novosibirsk, Russia

Abstract. Process-oriented programming is one of the approaches 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 the 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 needed for proving.

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. First we introduce the concept of basic patterns. Then we propose methods of generation of extra invariants patterns, lemmas and scripts for proving these lemmas for derived requirement patterns. Next we demonstrate our approach by an example. We also analyzed related works. The proposed approach allows us to make the deductive verification of process oriented programs more automated.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

Process-oriented programming [28] is a promising method for developing control software. This programming paradigm extends general-purpose imperative languages by integrating the concept of finite state machines and allows a program

* Supported by organization x.

to be described as a collection of interacting processes, which is particularly useful in the context of control software development. 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, called *STOP*, and the error stop state, called *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 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, whether it is in the state *STOP* or *ERROR*. When starting, the 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 is a process-oriented language that extends Structured Text (ST) language from the IEC 61131-3 standard [8]. 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, local variables `VAR`. A process definition contains declarations of variables and a sequence of state definitions. A state body consists of ST statements and poST-specific statements.

Control software requires formal verification because it has high reliability requirements. Deductive verification [12] is one of the formal verification methods in which the verification process is based on logical inference. In this method, 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 tue 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 [1]. When describing requirements a program is considered as a black box, i. e. the requirements express relations between input and output variables, but they 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 [4] 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. A number of methods was proposed for loop invariant synthesis. One of these method is the template-based method in which an invariant template defining form of the invariant and containing parameters is specified. Then the problem of determining the loop invariant is reduced to finding the values of these parameters and substituting these values into the template to obtain the invariant.

Earlier we developed a set of temporal requirement patterns in our language DV-TRL. It was noted that extra invariants needed for proving verification conditions for requirements satisfying the same pattern are similar and can also be described using patterns. In [5], we presented a set of the extra invariants patterns corresponding to the requirements patterns. In addition to the extra invariant pattern, lemmas and scripts for proving verification conditions in Isabelle/HOL [21] proof assistant are defined for each requirement pattern. However there are requirements that does not satisfy previously developed patterns. We are currently developing a requirement pattern system including basic patterns and a technique of their combining that wll allow one to describe the requirements by combining the basic patterns. This paper presents this technique including constructing the extra invariant pattern and the lemmas for a derived requirement patterns defined using the basic patterns combination as well as proving these lemmas.

This paper has the following structure. Section 3 presents schemes of basic patterns of requirements and extra invariants as well as patterns of lemmas defined for them. Section 5 gives general information about the derived requirements patterns and introduces the method of constructing the extra invariant pattern and algorithms for generation of lemmas and scripts for proving them corresponding to a derived requirement pattern. Section 6 gives an an example of a poST program and a requirement for this program. A derived requirement pattern that the requirement satisfy is presented, and the corresponding extra invariant pattern and lemmas are defined. Section 7 discusses related works on automated loop invariant generation. Section 8 summarizes the results and discusses the future work.

2 Approach to Automation of Deductive Verification

This section describes our approach to automation of deductive verification of process-oriented programs based on patterns. In this approach, a set of basic requirements patterns are defined. For each basic requirement pattern, a corresponding basic extra invariant pattern is defined. For each pair consisting of a basic requirement pattern and the corresponding basic extra invariant pattern, a set of lemmas used in proving formulas containing instances of these patterns are

defined. To specify requirements, derived requirements patterns are constructed by combining basic requirement patterns through conjunction, disjunction and pattern composition (nesting). Each derived requirement pattern has parameters and describes some class of requirements that require similar extra invariants and similar proofs. To set a requirement, a user needs to select or define the appropriate pattern and specify its parameters. For each derived requirement pattern, a corresponding derived extra invariant pattern is constructed from basic extra invariants patterns and lemmas are defined and proved using lemmas for basic patterns. The derived extra invariants patterns are used to specify requirement-dependent extra invariants. A derived extra invariant pattern contains parameters that correspond to the requirement pattern parameters and additional parameters. The values of the former are not always equal to the values of the corresponding parameters of the requirement pattern, but when specifying a specific requirement, when the parameter values do not contain nested patterns, their values are equal. The lemmas for derived patterns are used in proving verification conditions. Moreover in our approach, requirement-independent extra invariants that express simpler program properties relating process states, values of variables and process timers in one point of the program. We also have patterns for the requirement-independent extra invariants. Thus, a control loop invariant is a conjunction of a requirement and an extra invariant, and the latter in turn is conjunction of requirement independent invariants and a requirement dependent invariant. For proving verification conditions, we define proof scripts depending on the verification condition and the requirement being proven, the corresponding extra invariants and the lemma required for the proof, which is chosen depending on whether the requirement or the extra invariant is proved and which pattern it satisfies.

Earlier we developed a verification condition generator that automates the task of verification condition generation for poST-programs. In this work, we have developed algorithm for constructing extra invariants 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 previously developed verification condition generator. User interaction with the verification tool is shown in Fig. 2.

Let's consider user interaction with the verification tool. First, the user determines requirement-independent extra invariants based on the process-oriented program being verified. Several invariants can be defined. To specify each invariant, the user selects a requirement-independent extra invariants pattern and specifies parameter values for it. After determining requirement-independent extra invariants, the following actions are performed for each requirement to be verified. The user selects an appropriate derived requirement pattern from the knowledge base of the verification system, if any, and specifies parameter values for the requirement pattern and the corresponding requirement-dependent extra invariant pattern. If there is no appropriate requirement pattern, the defines it by combining basic and existing derived requirements patterns. Then using the pattern definition, the verification tool generates the corresponding derived extra-

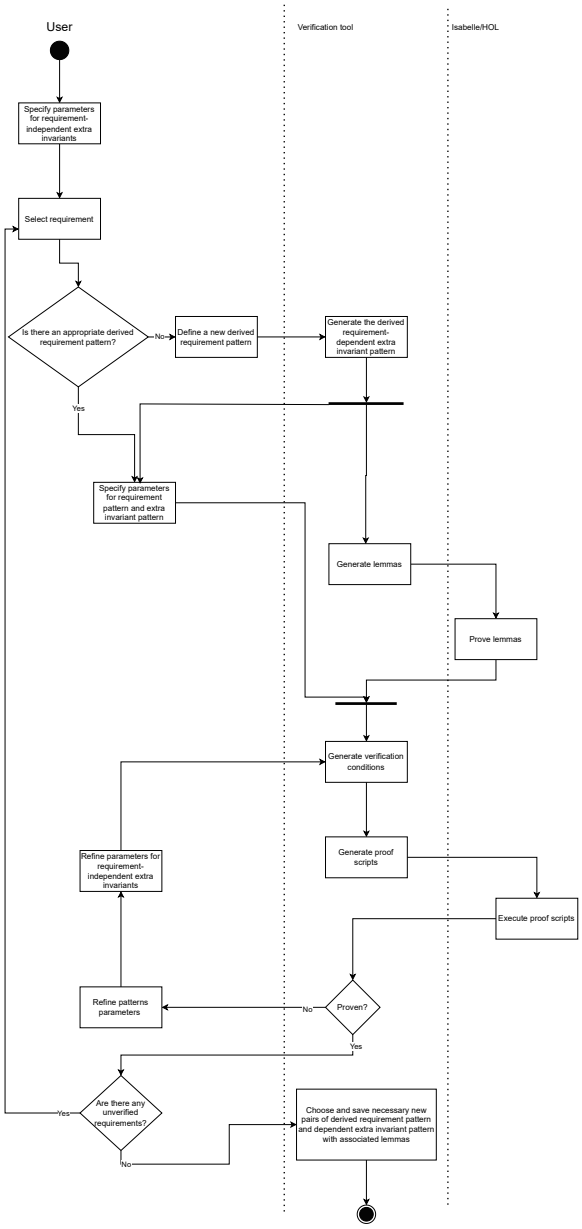

Fig. 1. A figure caption is always placed below the illustration. Please note that short captions are centered, while long ones are justified by the macro package automatically.

invariants pattern. Next, the user specifies parameter values for these patterns, and the verification tool generates the lemmas for the patterns that are proved in Isabelle/HOL proof assistant. 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 for the requirement and the extra invariants, and the verification tool proceeds to re-generation of verification conditions. If there are no unverified requirements, the verification tool saves necessary pair of derived requirements and extra invariants patterns with associated lemmas defined during the verification, and the verification of the program is completed.

3 Basic Extra Invariant Patterns and Lemmas

This section presents schemes of basic patterns of requirements and extra invariants and patterns of lemmas defined for these patterns as well as examples of basic patterns and the corresponding lemmas. Basic requirements extra invariants patterns are used for specifying requirements and extra invariants respectively and defining derived requirements patterns and derived extra invariants patterns corresponding to the derived requirement patterns. Basic lemmas are used for proving lemmas for derived patterns which in turn are used to prove verification conditions generated for requirements and extra invariants corresponding to these derived patterns. To talk about the truth of a temporal formula of language DV-TRL in a certain point in time, we present the temporal formula as a predicate with one argument s_1 of the update state date type. We say that a temporal formula A is true in an update state s_1 if $A(s_1)$ is true in the first-order logic. In particular, each control loop invariant is parameterized by an update state S. We prove that a control loop invariant is true in the update state before entering the loop and if the invariant is true in an external update state before an iteration, it is true in the update state before the iteration. Basic requirement patterns, and therefore basic extra invariant patterns are divided into future-time and past-time patterns. A formula A is a formula about the future if it describes an assertion about update states s_2 such that $s_1 \leq s_2$, where s_1 is the argument of A. A formula A is a formula about the past if it describes an assertion about update states s_2 such that $s_2 \leq s_1$. A future-time pattern describes some class of formulas about the future. A past-time pattern describes some class of formulas about the past.

The following designations are used in the patterns discussed below:

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

- $p(s)$ 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 $s_1 \leq s_2 \wedge \ldots \wedge s_{n-1} \leq s_n$;
- $e(s)$ 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;
- $consecutive(s_1, s_2)$ returns true if $e(s_1) \wedge e(s_2) \wedge s_1 \leq s_2 \wedge n(s_1, s_2) = 1$.

3.1 Future-Time Patterns

We specify a temporal requirement as a control loop invariant that is formula describing the history of variables values up to the update state s that is the parameter of the loop invariant. Therefore we cannot use variables values in update states that are not substates of the state s. Hence the future in our temporal requirements is bounded by the state s, and each future-time pattern is parameterized by two update states: s_1 (the initial state that is a parameter of a temporal formula specified using this pattern) and s (the final state that is a parameter of the loop invariant). The definition of a future-time 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 variables of the update state data type bound by quantifiers in R' such that $s_1 \leq r_{j_i} \leq s$ and A_1,..., A_n are formula parameters whose values are parameterized formulas defined as follows: 1) atomic parameterized formulas and their negations are parameterized formulas, 2) if A_1 and A_2 are parameterized 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 parameterized 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 parameterized formulas, then $\lambda(s, s_1).P(s, s_1, p_1, ..., p_{m'}, A_1, ..., A_{n'})$ is a parameterized formula. Since the update states r_{k_i} (i=1,..., n) in the pattern definition in which program variables values are described by A_i are such that $s_1 \leq r_{j_i}$ and s_1 is the update state corresponding to the present moment of time while s is just the upper bound of the considered future, formulas specified by the pattern are formulas about the future. In temporal logics, formulas can be transformed into the positive normal form [6] (chapter 4) in which negations are only applied to atomic formulas, given the dual operators. In our work, it is convenient to use formulas in the positive normal form so that not to develop general rules of constructing patterns used in extra invariants and lemmas for negations of arbitrary formulas. As in the positive normal form, only conjunction, disjunction and negation are used as logical connectives, with negation applied only to atomic formulas, but unlike the positive normal form in which only a limited set of temporal operators is used, in our formulas, any basic patterns are allowed.

Therefore we consider only combining of requirement patterns using conjunction and disjunction and nesting patterns.

For each requirement pattern, corresponding extra invariant pattern I is defined. Its definition has the following form: $I \equiv \lambda s.\lambda(s_1, p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n).I'(s_1, s, p_1, ..., p_m, ep_1($ where p_1,..., p_m are constant parameters whose values are coincide with the values of the corresponding parameters of the basic requirement pattern, ep_1,..., ep_k are parameters-functions, I' 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 variables of the update state data type bound by quantifiers in I', such that $s_1 \leq r_{j_i} \leq s$ and A_1,..., A_n are formula parameters whose values are parameterized formulas defined as follows: 1) atomic parameterized formulas and their negations are parameterized formulas, 2) if A_1 and A_2 are parameterized 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 parameterized formulas, 3) if P is a future-time extra invariant pattern with m' constant parameters, k' parameters-functions and n' formula parameters, p_1,..., $p_{m'}$ are constants of the appropriate types, ep_1,..., $ep_{k'}$ are functions of the appropriate types and A_1,..., $A_{n'}$ are parameterized formulas, then $\lambda(s, s_1).P(s, s_1, p_1, ..., p_{m'}, ep_1, ..., ep_{k'}, A_1, ..., A_{n'})$ is a parameterized formula, 4) if P is a past-time 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, then $\lambda(s, s_1).P(s_1, s, p_1, ..., p_{m'}, A_1, ..., A_{n'})$ is a parameterized formula. Values of the parameters A'_i $(i = 1, ..., n)$ are not the same as the values of the corresponding parameters A_i of the requirement pattern in general, but they are related and coincide if they do not contain nested future-time patterns. For each future-time pattern, lemmas are created according to the following patterns:

1. LP_1:

$$
\begin{array}{l}
consecutive(s, s') \longrightarrow \\
cond_{true}(p_1, ..., p_m) \vee \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_1(s, s_1) \longrightarrow A'_1(s', s_1)) \wedge ... \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_n(s, s_1) \longrightarrow A'_n(s', s_1)) \wedge \\
cond(p_1, ..., p_m, ep_1(s), ..., ep_k(s), ep_1(s'), ..., ep_k(s'), A'_1(s'), ..., A'_n(s')) \longrightarrow \\
\forall s_1.e(s_1) \wedge s_1 \leq s \wedge I(s)(s_1, p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n) \longrightarrow I(s')(s_1, p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n)
\end{array}
$$

Lemmas satisfying this pattern are used to prove lemmas applied for proving verification conditions for extra invariants. Here and in other lemmas relating extra invariants before and after a control loop iteration, the variable s denotes an update state before the iteration, and s' denotes the update state after the iteration. The premise of a lemma satisfying this pattern states that if the condition $cond_{true}$, under which the requirement is identically true, for example, the equality of the value of a time parameter to zero, is false, then for each update state up to s, if a formula $A'_i(s)$ is fulfilled, then the formula $A'_i(s')$ is fulfilled in the same state, $i = 1, ..., n$ and the formula $cond$ that expresses relation between $A'_i(s', s')$, p_i, $ep_i(s)$ and $ep_i(s')$ is fulfilled. The

conclusion of the lemma states that if the subformula of the extra invariant with an initial state s_1 and the final state s was fulfilled, then the subformula of the extra invariant with the final state s' is also fulfilled. Note that the conclusion of the lemma is similar to the premises, where $I(s_1, s, p, ep, A_1, ..., A_n)$ is used instead of A_i, and $op_inv(s_1, s', p, ep', A'_1, ..., A'_n)$ is used instead of A_i. This allows defining and proving lemmas for derived patterns by induction on the construction of this pattern.

2. LP_2

$$e(s) \longrightarrow cond(p_1, ..., p_m, ep_1(s), ..., ep_k(s), A'_1(s), ..., A'_n(s)) \longrightarrow I(s, s, p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n)$$

Lemmas satisfying this pattern are used in proofs of lemmas applied for proving verification conditions for extra invariants, in the case when the initial and the final update states coincide. Condition *cond* express a relation between $A'_1(s, s)$,..., $A'_n(s, s)$, p_1,..., p_m, $ep_1(s)$,..., $ep_k(s)$.

3. LP_3:

$$\begin{array}{l}(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_1(s, s_1) \longrightarrow A_1(s, s_1)) \wedge ... \wedge \\ (\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_n(s, s_1) \longrightarrow A_n(s, s_1)) \wedge \\ cond(p_1, ..., p_m, ep_1(s), ..., ep_k(s)) \longrightarrow \\ \forall s_1.e(s_1) \wedge s_1 \leq s \wedge I(s, s_1, p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n) \longrightarrow R(s, s_1, p_1, ..., p_m, A1, ..., An)\end{array}$$

Lemmas satisfying this pattern are used to prove lemmas applied for proving verification conditions for requirements using extra invariants. The premise *cond* of lemmas satisfying this pattern relates the values of $p_1, ..., p_m$ and $ep_1(s)$,..., $ep_k(s)$.

values of the parameters $cond_{true}$ and *cond* in lemmas satisfying the lema patterns LP_1, LP_2 and LP_3 are quantifier-free formulas that do not contain patterns.

Next, let's consider the future-time patterns. For each requirement pattern, the extra invariant pattern is also given, as well as the conditions $cond_{true}$, $cond_1$, $cond_2$ and $cond_3$ from lemmas 1—3.

Let's consider an example of a future-time requirement pattern. This requirement pattern 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. This pattern has the following parameters:

- constant parameters: t;
- formula parameters: A_1, A_2

The formula R' is defined as follows:

$$\begin{array}{l}R'(s_1, s, t, A_1, A_2) \equiv \\ n(s_1, s) \geq t \longrightarrow \\ (\exists r_2.e(r_2) \wedge s_1 \leq r_2 \leq s \wedge n(s_1, r_2) \leq t \wedge A_2(r_2) \wedge \\ \quad (\forall r_1.(e(r_1) \wedge s_1 \leq r_1 \leq r_2 \wedge r_1 \neq r_2 \longrightarrow A_1(r_1)))),\end{array}$$

The corresponding extra invariant pattern has one parameter-function t_1. The formula I' used in the definition of the extra invariant pattern is defined as follows:

$$
\begin{array}{l}
I'(s_1, s, t, t_1, A_1, A_2) \\
(\exists r_2.e(r_2) \wedge s_1 \leq r_2 \wedge r_2 \leq s \wedge n(s_1, r_2) \leq t \wedge A_2(r_2) \wedge \\
\quad (\forall r_1.e(r_1) \wedge s_1 \leq r_1 \wedge r_1 < r_2 \longrightarrow A_1(r_1))) \vee \\
n(s_1, s) < t_1 \wedge \\
(\forall r_1.e(r_1) \wedge s_1 \leq r_1 \wedge r_1 \leq s \longrightarrow A_1(r_1))
\end{array}
$$

This pattern contains one additional parameter t_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 if time t_1 has passed from the state s_1 to the final state s, then event A_2 has occurred even if time t has not passed. In addition If time t_1 has not yet passed after the start of the time counting in state s_1, then condition A_1 must be fulfilled. The value of the parameter t_1 determines the maximum waiting time for the event A_2 in the state s.

Conditions in lemmas for this pattern are following:

$$
cond_{true} \equiv False\,;\;\; cond_{LP_1} \equiv t_1 = 0 \vee A_2'(s) \wedge t_1 \leq t \vee A_1'(s) \wedge t_1 < t_1'\,; \\
cond_{LP_2} \equiv A_2(s) \vee A_1(s) \wedge t1 > 0\,;\;\; cond_{LP_3} \equiv t_1 \leq t
$$

The condition $cond_{true} = False$ indicates that there is no such value of t at which the requirement and the extra invariant patterns are identically true for all A_1, A_2. The condition $cond_{LP_1}$ asserts that one of the following conditions is satisfied: 1) the value of the additional parameter t_1 in the invariant before the iteration of the loop is zero, which means that event A_2 occurred no later than when the program was in the update state s, 2) event A_2' occurred in the update state s' and the waiting time for the event A_2 has not been exceeded, or 3) condition A_1 is satisfied in the update state s and the waiting time has increased by at least 1. In the second lemma, the condition $cond_{LP_2}$ asserts that the event A_2 has occurred or the condition A_1 is true and the event A_2 is expected in the future. In the third lemma, the condition $cond_{LP_3}$ asserts that the maximum waiting time does not exceed the value of the parameter t.

3.2 Past-Time Patterns

Since instances of past-time patterns can contain nested future-time patterns, each past-time pattern is also parameterized by two update states: s_1 (a parameter of a temporal formula specified using this pattern) and s (a parameter of the loop invariant). The definition of a past-time 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 r_{j_i} $(i = 1, ..., n)$ are variables of the update state data type bound by quantifiers in R' such that $r_{j_i} \leq s_1$, other parameters have the same meaning as in the future-time patterns. Since the update states r_{k_i} (i=1,..., n) in the pattern definition in which program variables values are described by A_i are such that $r_{j_i} \leq s_1$ and s_1 is the update state corresponding to the present moment of time, formulas specified by the pattern are formulas about the past.

For each past-time requirement pattern, corresponding extra invariant pattern I is defined. Its definition has the following form: $I \equiv \lambda s.\lambda(p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n).I'(s, p_1, ..., p_m, r_{j_1}, ..., r_{j_n}, ep_1, ..., ep_k, A'_1, ..., A'_n)$, where r_{j_i} $(i = 1, ..., n)$ are variables of the update state data type bound by quantifiers in I' such that $r_{j_i} \leq s$, other parameters have the same meaning as in the future-time patterns.

For each past-time pattern, lemmas are created according to the following patterns:

1. LP_1

$$
\begin{array}{l}
consecutive(s, s') \Longrightarrow \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_1(s, s_1) \longrightarrow A'_1(s's_1)) \wedge ... \wedge \\
(\forall s_1.e(s_1) \wedge s_1 \leq swedge A'_n(s, s_1) \Longrightarrow A'_n(s', s_1)) \Longrightarrow \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge R(s, s_1, p_1, ..., p_m, A'_1, ..., A'_n) \longrightarrow R(s', s_1, p_1, ..., p_m, A'_1, ..., A'_n)
\end{array}
$$

   Lemmas satisfying this pattern are used to prove lemmas applied for proving verification conditions for extra invariants. The premise of this lemma states that, for each update state up to s, if the formula $A'_i(s)$ $(i = 1, ..., n)$ is fulfilled, then $A'_i(s')$ is fulfilled in the same state, The conclusion of the lemma states that if a subformula of an extra invariant satisfying in the state s was fulfilled, then the formula of the extra invariant satisfied in the state s' is fulfilled in the same state. Note that the conclusion of the lemma is similar to the premises, where $R(s, s_1, p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n)$ is used instead of A_i, and $R(s, s_1, p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n)$ is used instead of A_i. This allows defining and proving lemmas for derived patterns by induction on the construction of this pattern.
2. LP_3

$$
\begin{array}{l}
e(s) \Longrightarrow \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_1(s, s_1) \longrightarrow A_1(s, s_1)) \wedge ... \wedge \\
(\forall s_1.e(s_1) \wedge s_1 \leq swedge A'_n(s, s_1) \Longrightarrow A_n(s, s_1)) \Longrightarrow \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge R(s, s_1, p_1, ..., p_m, A'_1, ..., A'_n) \longrightarrow R(s, s_1, p_1, ..., p_m, A_1, ..., A_n)
\end{array}
$$

   This pattern is similar to the previous one, but lemmas satisfying this pattern are used to prove lemmas applied for proving verification conditions for requirements.
3. L_4

$$
\begin{array}{l}
consecutive(s, s') \longrightarrow \\
I(s, p_1, ..., p_m, ep_1, ep_k, A'_1, ..., A'_n) \wedge \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_1(s, s_1) \longrightarrow A'_1(s', s_1)) \wedge ... \wedge \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_n(s, s_1) \longrightarrow A'_n(s', s_1)) \wedge \\
cond(p_1, ..., p_m, ep_1(s), ..., ep_k(s), ep_1(s'), ..., ep_k(s'), A_1(s', s'), ..., A_n(s', s') \Longrightarrow \\
I(s', p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n)
\end{array}
$$

Lemmas satisfying used to prove the additional conjunct of the extra invariant in the proofs of lemmas used to prove the verification conditions for extra invariants. The premise of this lemma states that, for each update state up to s, if a formula $A'_i(s)$ $(i = 1, ..., n)$ is fulfilled, then $A'_i(s')$ is fulfilled in the same state, *cond* is some formula relating $A'_i(s', s')$, p_1,..., p_m, $ep_1(s)$,..., $ep_k(s)$ and $ep_1(s')$,..., $ep_k(s')$, $I(s, p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n)$ that is an additional conjunct of an extra invariant that is fulfilled at the beginning of the iteration of the loop. The conclusion of the lemma is the additional conjunct of the extra invariant that is fulfilled at the end of the iteration in the state s'.

4. L_5

$$
\begin{array}{l}
e(s) \longrightarrow \\
I(s, p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n) \wedge \\
cond(p_1, ..., p_m, ep_1(s), ..., ep_k(s) \Longrightarrow \\
R(s, s, p_1, ..., p_m, A'_1, ..., A'_n)
\end{array}
$$

Lemmas satisfying this pattern are used to prove lemmas applied for proving verification conditions for an extra invariant in the case where it is necessary to prove that the subformula of the extra invariant satisfying a past-time requirement pattern is fulfilled in the state s that is a parameter of the loop invariant, using the additional conjunct of the extra invariant. The premise of this lemma states that some formula *cond* relating p_1,..., p_m and $ep_1(s)$,..., $ep_k(s)$ is true, an additional conjunct of an extra invariant $I(s, s, p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n)$ is fulfilled in the state s. The conclusion of the lemma is a part of an extra invariant satisfying a past-time requirement pattern that is fulfilled at the same iteration in the state s.

5. LP_8

$$
\begin{array}{l}
consecutive(s, s') \longrightarrow \\
I(s, p_1, ..., p_m, ep_1, ..., ep_k, A'_1, ..., A'_n) \wedge \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_1(s, s_1) \longrightarrow A'_1(s', s_1)) \wedge ... \wedge \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_n(s, s_1) \longrightarrow A'_n(s', s_1)) \wedge \\
cond(p_1, ..., p_m, ep_1(s), ..., ep_k(s), ep_1(s'), ..., ep_k(s'), A'_1(s', s'), ..., A'_n(s', s')) \Longrightarrow \\
R(s', s', p_1, ..., p_m, A'_1, ..., A'_n)
\end{array}
$$

This pattern is similar to the pattern L_5, but the instance of an extra invariant pattern in the premise is satisfied at the previous, rather than the same iteration as the requirement pattern instance in the conclusion. The condition *cond* relates $A'_i(s', s')$, p_1,..., p_m, $ep_1(s)$,..., $ep_k(s)$ and $ep(s')$.

A lemma satisfying the pattern LP_5 does not have to be defined for a past-time pattern because it is used as an auxiliary lemma for defining and proving a lemma satisfying the pattern LP_8. For some past-time patterns, the lemma satisfying LP_5 cannot be defined. But for some other patterns, it was easier to prove lemma satisfying LP_5 and use it to define and prove a lemma satisfying LP_8. Moreover requirement patterns can be used as past-time patterns because

they express assertions about the values of variables in update states preceding the current one in which the control loop invariant holds. The lemmas satisfying LP_4 and LP_5 are defined for requirement patterns, but they need to be transformed to the form of the lemmas for past-time patterns. Given the lemmas satisfying LP_4 and LP_5, the condition *cond* of the lemma satisfying LP_8 can be constructed automatically. The condition *cond* in the lemma satisfying LP_8 is the conjunction of the conditions $cond - LP_4$ and the formula obtained from $cond_{LP_5}$ by replacing s with s'.

Let's consider an example of a past-time requirement pattern `always`. This pattern describes requirements that state that a condition A_1 should always be true between iterations of the control loop up to the current state s_1. This pattern has one formula parameter A_1. The formula R' is defined as follows:

$$R'(s, s_1, A_1) \equiv$$
$$\forall r_1.e(r_1) \wedge r_1 \leq s_1 \longrightarrow A_1(s, r_1)$$

The corresponding extra invariant pattern `always_inv` coincides with the requirement pattern when $s_1 = s$, i. e., the formula I' is defined as follows:

$$I'(s, A_1') \equiv$$
$$\forall r_1.e(r_1) \wedge r_1 \leq s \longrightarrow A_1'(s, r_1)$$

If this pattern is used in requirements only as principal operator, that is, requirements have the form $always(s, s_1 A_1)$, where A_1 does not contain an instance of the pattern `always`, then the lemma satisfying the pattern LP_1 does not need to be defined. In addition, since the requirement and extra invariant patterns coincide, lemmas satisfying the patterns LP_4 and LP_8 also coincide. The condition *cond* in the lemma satisfying LP_4 is $A_1'(s', s')$ Since this pattern is used as an outer requirement pattern when constructing derived patterns, the lemma L_4 is transformed to the form of the corresponding lemma for derived requirement patterns and has the form:

$$always\_inv(s, A_1') \longrightarrow$$
$$consecutive(s, s') \longrightarrow$$
$$(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A_1'(s, s_1) \longrightarrow A_1'(s', s_1)) \wedge$$
$$A_1'(s', s') \longrightarrow$$
$$always\_inv(s', A_1')$$

The condition *cond* in the lemma satisfying the pattern LP_5 is $True$ because the requirement pattern for $s_1 = s$ and the extra invariant patterns coincide and for an extra invariant to imply the requirement, it is sufficient that the value of the parameter A in the extra invariant imply the value of the parameter A in the requirement.

4 Derived Requirements and Extra Invariants Patterns

This section describes the schemes of patterns that are directly used to specify requirements and extra invariants and can be used as outer patterns in definitions

of more complex derived requirements patterns as well as patterns of lemmas that are defined for the derived requirements and extra invariants patterns. A past-time requirement pattern for $s_1 = s$ can be considered as a particular case of a derived requirement pattern. The extra invariant pattern corresponding to it is the basic extra invariant pattern corresponding to it as a basic pattern.

A definition of a requirement pattern that is not a basic 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 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 $\lambda s_1.A_1(s_1) \vee A_2(s_1)$; 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 requirement parameter formulas, s is an update state. 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. For example, we allow applying negations to atomic formulas containing formula parameters. Values of these parameters cannot contain nested patterns because otherwise, the formula would not be in the positive normal form, and bringing it to this form would lead to the fact that the outer pattern would not be used. Let 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, and therefore they do not depend on s.

For each derived requirement pattern the corresponding extra invariant pattern. A definition of a derived extra invariant pattern that is not a basic pattern has the following form: $I \equiv \lambda s.\lambda(p_1, ..., p_m, ep_1, ..., ep_k, b_1, ..., b_l, A'_1, ..., A'_n).I'(s, p_1, ..., p_m, ep_1(s), ..., ep_k(s)b_1(s)$ where s is the update state in which the extra invariant should be true; p_1,..., p_m are constant parameters whose values in an extra invariant are the same as in the corresponding requirement; b_1,..., b_l are parameters-functions returning boolean values and introduced for nested past-time patterns; ep_1,..., ep_k are other parameters-functions corresponding to parameters-functions of extra invariant pattern instances included in I'; A'_{j_1},..., A'_{j_v} are formula parameters whose values are extra invariant parameter formulas; A'_{h_1},..., A'_{h_u} are formula parameters whose values are pattern parameter formulas as in the requirement pattern, $A'_{h_i} \equiv A_{h_i}$ ($i = 1, ..., u$); I' is a parameterized formula that has the form $P(s, p_1, ..., p_{m'}, ep_1, ..., ep_{k'}, A_1, ..., A_{n'}) \wedge (b_1 \longrightarrow I_1) \wedge ... \wedge I_l$ where P is a derived extra invariant pattern, A_1,..., $A_{n'}$ are parameterized extra invariant parameter formulas containing parameterized past-time requirement pattern instances R_1,..., R_l, I_1,..., I_l are parameterized extra invariants corresponding to R_1,..., R_l respectively, $b_1(s)$,..., $b_l(s)$ indicate whether the invariants I_1, I_l should be true in the state s.

For each derived pattern, 4 lemmas are defined. The first two lemmas satisfy the following patterns:

1. LP_4

$$I(s, p_1, ..., p_m, ep_1, ..., ep_k, b_1, ..., b_l, A'_1, ..., A'_n) \longrightarrow$$
$$consecutive(s, s') \longrightarrow$$
$$(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_{j1}(s, s_1) \longrightarrow A'_{j1}(s', s_1)) \wedge ... \wedge$$
$$(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_{jv}(s, s_1) \longrightarrow A'_{jv}(s', s_1)) \wedge$$
$$cond(p_1, ..., p_m, ep_1(s), ..., ep_k(s), b_1(s), ..., b_l(s), ep_1(s'), ..., ep_k(s'), b_1(s'), ..., b_l(s'), A'_{h_1}(s'), ..., A'_{h_u}(s'), A'_{j1}(s', s'), ...$$

$$I(s', p_1, ..., p_m, ep_1, ..., ep_k, b_1, ..., b_l, A'_1, ..., A'_n)$$

The premises of a lemma satisfying this pattern state that an extra invariant is fulfilled at the beginning of the iteration of the control loop in the state s, for all update states up to s, if the formula $A'_{j_i}(s)$ $(i = 1, ..., v)$ is fulfilled, then the formula $A'_{j_i}(s')$ is fulfilled in the same state and the condition *cond* relating $A'_{h_i}(s')$, $A'_{j_i}(s', s')$ and the values of the parameters p_1,..., p_m, $ep_1(s)$,..., $ep_k(s)$, $b_1(s)$,..., $b_l(s)$, $ep_1(s')$,..., $ep_k(s')$ and $b_1(s')$,..., $b_l(s')$ is fulfilled. The conclusion of the lemma states that the extra invariant is satisfied at the end of the iteration in state s'.

2. LP_5

$$I(s, p_1, ..., p_m, ep_1, ..., ep_k, b_1, ..., b_l A'_1, ..., A'_n) \longrightarrow$$
$$e(s) \longrightarrow$$
$$(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_{j1}(s, s_1) \longrightarrow A'_{j1}(s', s_1)) \wedge ... \wedge$$
$$(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_{jv}(s, s_1) \longrightarrow A_{jv}(s, s_1)) \wedge$$
$$cond(p_1, ..., p_m, ep_1(s), ..., ep_k(s), b_1(s), ..., b_1(s)) \longrightarrow$$
$$R(s, p_1, ..., p_m, A_1, ..., A_n)$$

The premises of a lemma satisfying this pattern state that an extra invariant is fulfilled in the state s, for all update states up to s, if the formula $A'_{j_i}(s)$ $(i = 1, ..., v)$ is fulfilled, then another formula $A_{j_i}(s)$ is fulfilled in the same state and the condition *cond* relating the values of the parameters p_1,..., p_m, $ep_1(s)$,..., $ep_k(s)$ and $b_1(s)$,..., $b_l(s)$ is also fulfilled. The conclusion of the lemma states that in the state s, the requirement is fulfilled.

We will call derived patterns considered above general patterns. They can be used to define new derived patterns. But in the case where formula parameters of a pattern do not contain nested patterns, the lemmas satisfying the patterns LP_4 and LP_5 can be simplified so that they do not contain formulas with quantifiers. Therefore for each general requirement or extra invariant pattern, we can define the corresponding particular pattern whose formula parameters cannot contain nested patterns.

5 Construction Of Extra Invariant Patterns and Lemmas for Derived Requirement Patterns

This section describes how to make derived requirement patterns and extra invariant patterns from basic patterns, as well as how to make lemmas for proving requirements and extra invariants and prove them.

The other two lemmas L_6 and L_7 are obtained from lemmas L_4 and L_5, respectively by identifying each A'_i with the corresponding A_i $(i = 1, ..., n)$ and excluding premises of the form $\forall s_1.(e(s_1) \wedge s_1 \leq s \wedge A_i(s_1) \longrightarrow A_i(s_1))$. Lemmas L_4 and L_5 are used in proofs of similar lemmas created for combinations of the requirement pattern R with other patterns and in proofs of the lemmas L_6 and L_7 respectively for the pattern R. Lemmas L_6 and L_7 are used to prove verification conditions. The lemmas L_6 and L_7 can be proven in Isabelle/HOL automatically using the proof method `auto`. In some cases, the lemma L_5 can be proven automatically using Sledgehammer, but manual proving is sometimes needed. To prove the lemmas L_4 and L_5, we define recursive functions $proveOuter(inv)$ and $prove()$ that generates a script for proving these lemmas. The function $proveOuter(inv)$ generates proof script for proving a goal that is a pattern instance or boolean combination of pattern instances that is a subformula of the conclusion of a lemma being proven and is not nested in other patterns. This function takes one argument inv is an extra invariant from the lemma premises or its part used for proving the current goal. The function $prove$ generates a proof script for proving formulas obtained during proving a lemma after applying some lemma.

Next, we describe the methodology for creating an extra invariant pattern, generating lemmas L_4 and L_5, as well as the functions $proveOuter$ and $prove$ for conjunction, disjunction, nesting a future-time pattern and nesting a past-time pattern.

5.1 Extra Invariants

An extra invariant pattern for a conjunction of requirement patterns is defined as a conjunction of the corresponding extra invariant patterns. An extra invariant pattern for the disjunction of requirement patterns is defined as the disjunction of the corresponding extra invariant patterns.

Let's consider a composition of patterns, where the value of a parameter A of the outer pattern is a boolean combination a of atomic formulas and nested instances of patterns (possibly consisting of a single pattern instance). The extra invariant pattern for this composition is obtained from the extra invariant pattern for the outer requirement pattern by the following procedure: 1) a boolean combination obtained from a by replacing each future-time pattern with the corresponding extra invariant pattern is substituted for A; 2) an additional conjunct $b \longrightarrow I$ for occurrence of a past-time pattern is added to the extra invariant, where b is a new boolean parameter of the extra invariant pattern for the composition and I is the extra invariant pattern corresponding to the past-time pattern.

5.2 Lemma Generation

This subsection describes the generation of lemmas L_4 and L_5 from this section. If the derived requirement pattern is a conjunction or disjunction of two patterns, then the conditions $cond_4$ and $cond_5$ in lemmas L_4 and L_5 are conjunctions of the corresponding conditions from the lemmas for each of the patterns.

In the case of pattern nesting, the lemma L_4 is defined by the following algorithm:

1. Take the lemma satisfying the lemma pattern LP_4 for the outer extra invariant pattern as L_4;
2. Replace extra invariants in the lemma by extra invariants satisfying the extra invariant pattern for the composition;
3. Let C be the premises of the lemma third L_4?;
4. Substitute the value of the parameters A and A' of the outer pattern instances containing nested patterns into C;
5. Add additional conjuncts of the extra invariant in the conclusion of the lemma as conjuncts to C;
6. While C contains nested patterns apply an appropriate rule from the following set:
   (a) Replace each subformula of the form $(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge (P(s_1) \wedge Q(s_1)) \longrightarrow (P'(s_1) \wedge Q'(s_1)))$ or $(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge (P(s_1) \vee Q(s_1)) \longrightarrow (P'(s_1) \vee Q'(s_1)))$ in C with the formula $(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge P(s_1) \longrightarrow P'(s_1)) \wedge (\forall s_1.e(s_1) \wedge s_1 \leq s \wedge Q(s_1) \longrightarrow Q'(s_1)$;
   (b) Replace each formula of the form $(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge P(s_1) \longrightarrow P'(s_1))$ in C, where P is a basic pattern, with the premises of the lemma L_1 for the pattern P;
   (c) If C contains a formula satisfying an instance of a future-time pattern in which the initial state coincides with the final state s', then replace this instance with the corresponding condition $cond_2$ from the lemma L_2 for this pattern;
   (d) If C contains a formula satisfying a past-time requirement pattern which is fulfilled at the end of the iteration in state s', replace this formula with the premises of the corresponding lemma L_8 for this pattern;
   (e) If C contains a formula satisfying a past-time extra invariant pattern which is fulfilled at the end of the iteration in state s', replace this formula with the premises of the corresponding lemma L_4 for this pattern;
   (f) If C contains a formula satisfying a past-time extra invariant pattern which is fulfilled at the beginning of the iteration in state s, replace this formula with the boolean parameter corresponding this occurrence of the past-time extra invariant pattern introduced in the previous subsection.
7. Return the obtained lemma L_4 as a result.

To ensure the unambiguity of the rule selection in the loop at step 6, the order of their application can be set.

The algorithm for generating the lemma L_5 is similar to the algorithm for generating the lemma L_4, but it has the following differences:

1. In step 1, the lemma L_5 for the outer pattern is used instead of the lemma L_4;
2. In step 2, the requirement pattern in the conclusion of the lemma is replaced with the requirement pattern for the composition;
3. The step 5 is not performed because the conclusion of the lemma does not contain an extra invariant;
4. In step 6b, the lemma L_3 is used instead of the lemma L_1 if P is a future-time pattern
5. The steps 6c, 6d, 6e and 6f are not performed because the condition $cond_5$ of a lemma L_5 does not contain parameters containing nested patterns.

5.3 Proving Lemmas

We define proof methods for proving lemmas. These proof methods have been implemented in Eisbach [20]. First, we define an auxiliary method `prove` that is used for proving formulas that are not subformulas of the conclusion of the lemma to be proven arise in proving after applying other lemmas. This method is defined as follows:

```
1  method prove declares simps =
2  (*formula without patterns or past-time pattern instance*)
3  (simp add: simps);fail |
4  (*conjunction*)
5  (match conclusion in "?P ∧ ?Q" ⇒
6    erule   conjE;
7  match premises in e[thin]: "_" ⇒
8    match   premises in p1[thin]: "_" ⇒
9    match   premises in p2[thin]: "_"       rule   conjI,(insert
       e p1)[1],prove,(insert e p2)[1],  p  r  o  v  e    ) |
10 (*disjunction*)
11 (match conclusion in "?P ∨ ?Q" ⇒   erule   disjE,rule disjI1,
       prove,rule disjI2,p r o v e ) |
12 (*always_imp(s, A, A)*)
13 (match conclusion in "always_imp ?s ?A ?A" ⇒   rule
      all_imp_refl ) |
14 (*pattern*)
15 rule,(assumption | simp),prove
```

First, this method ties to prove the goal by method `simp` (line 3). Formulas that does not contain nested patterns and instances of past-time patterns in the state s (before the iteration) are proved by the method `simp`. If the goal cannot be completely proved by themethod `simp`, it is left unchanged and other alternatives are tried in the following order.

If the conclusion of the current goal is a conjunction, the goal contains two premises: the first is $toEnvPs$ or $consecutive(s, s')$ and the second is a conjunction. In this case, the conjunction in the premise is eliminated. Then the conjunction in the conclusion is split. To prove each conjunct, the first premise e and the corresponding conjunct of the second premise (respectively $p1$ and $p2$

for the first and the second conjunct) of the original goal. The method `prove` is recursively invoked to prove each conjunct.

If the conclusion of the current goal is a disjunction, the goal contains two premises: the first is $toEnvPs$ or $consecutive(s, s')$ and the second is a disjunction. In this case, the disjunction in the premise is eliminated. Then the corresponding disjunct in the conclusion is proved, depending on which disjunct in the premise is true. The method `prove` is recursively invoked to prove each subgoal.

If the conclusion of the current goal is an instance of the pattern $always_imp$ and the values of the parameters A and A' of this pattern are the same, the lemma $always\_imp\_refl$ is used as an introduction rule. Since the lemma does not have premises, the application of the lemma proves the current goal completely.

In other cases, the conclusion of the current goal is a pattern instance. In this case, the appropriate lemma having the attribute `intro` is applied as an introduction rule. The application of the lemma produces two subgoals. The first subgoal is $e(s)$, $e(s')$ or $consecutive(s, s')$ (in the second case, the subgoal may contain schematic variables). To prove this subgoal, we first try the method `assumption`. The method `assumption` fails only if the conclusion of the goal is $e(s')$ and the goal contains a premise $consecutive(s, s')$. This goal does not contain schematic variables and is proved by the method `simp`. Then the method `prove` is recursively invoked to prove the second subgoal.

Then we define the main method `proveOuter` that is used for proving lemmas L_4 and L_5. This method is defined as follows:

```
method proveOuter declares simps =
(*additional conjunct of extra invariant*)
assumption |
(*conjunction *)
(match conclusion in "?P wedge ?Q" Rightarrow
  erule   conjE;erule conjE;
match premises in e[thin]: "_" Rightarrow
  match   premises in i1[thin]: "_" Rightarrow
  match   premises in i2[thin, simp]: "_" Rightarrow
  match   premises in p1[thin]: "_" Rightarrow
  match   premises in p2[thin]: "_" Rightarrow
  rule   conjI,(insert e i1 p1)[1],proveOuter,(insert e i2 p2)
    [1], p r o v e O u t e r   ) |
(*disjunction*)
(match conclusion in "?P ∨ ?Q" Rightarrow
  erule   conjE;
match premises in p1[thin]: "_" Rightarrow
  match   premises in p2[thin]: "_" Rightarrow
  match   premises in p3[thin]: "_" Rightarrow
  match   premises in p4[thin]: "_" Rightarrow
    (insert p1 p2)[1],erule disjE,(insert p3)[1],rule disjI1,
   proveOuter,(insert p4)[1],rule disjI2,
    p r o v e O u t e r   ) |
```

```
(*pattern*)
(match premises in p1[thin]: "_" (cut) Rightarrow
  match   premises in p2[thin]: "_" (cut) Rightarrow
  match   premises in p3[thin]: "_" (cut) Rightarrow
    (insert p1 p2)[1],(simp only: conj_assoc)?,(erule conjE)
   ?,
match premises in q1[thin]: "_" (cut) Rightarrow
  match   premises in q2[thin]: "_" (cut) Rightarrow
   (match premises in q3[thin]: "_" (cut) Rightarrow
       succeed   )?,(insert q1 q2)[1],erule elims,assumption
   ,(insert p3)[1],
(prove simps: simps)               )
```

First, this

6 Example

In this section, we consider an example of constructing a derived requirement and the corresponding extra invariant pattern and lemmas for proving verification conditions for requirements and extra invariants satisfying these patterns.

Let's 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 the following listing:

```
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
```

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`.

Let's consider the following requirement for the hand dryer control program: "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".

This requirement satisfies the following derived requirement pattern: "If event A_1 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 has one constant parameter t and three formula parameters A_1, A_2 and A_3. The formula R' for this pattern is defined as follows

$$R'(s,t,A_1,A_2,A_3) \equiv$$
$$always(s,s,(\lambda r_2 r_1.\neg A_1(r_1) \vee constrained\_until(r_2,r_1,t,A_2A_3)))\,.$$

In this pattern, negation is applied to the parameter A_1. Therefore the value of this parameter cannot contain nested patterns.

The corresponding extra invariant pattern has one extra parameter t_1. The formula I' is defined as follows:

$$I'(s,t,t_1,A_1,A_2,A3) \equiv$$
$$always\_inv(s,(r_2 r_1.\neg A_1(s_1) \vee constrained\_until\_inv(r_2,r_1,t,t_1,A_2,A3)))$$

Let's consider how this extra invariant pattern was constructed. The principal operator of the requirement pattern is `always`. Therefore the principal operator of the derived requirement pattern is the extra invariant pattern corresponding to the requirement pattern `always` that is `always_inv`. The value of the parameter A_1 of the pattern `always` in the derived requirement pattern is a disjunction. Hence the value of the parameter A_1 of the extra invariant pattern is also a disjunction. The left disjunct in the requirement pattern is a parameter with negation. It is substituted as the left disjunct into the extra invariant pattern unchanged. The right disjunct in the requirement pattern is an instance of the future-time pattern `constrained_until`. Consequently the right disjunct in the extra invariant pattern is a corresponding instance of the pattern `constrained_until_inv`. The values of the formula parameters

of the pattern `constrained_until` are parameters of the derived requirement pattern. They are substituted as the corresponding parameters of the pattern `constrained_until_inv` unchanged.

Let's describe constructing the lemma satisfying the lemma pattern LP_4 for this derived requirement pattern according the algorithm.

1. The requirement pattern is a composition. Therefore first we take the lemma satisfying the lemma pattern LP_4 for the outer pattern `always` as the lemma L_4. It has the following form:

$$always\_inv(s, A'_1) \longrightarrow consecutive(s, s') \longrightarrow (\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_1(s, s_1) \longrightarrow A'_1(s', s_1)) \wedge A'_1(s', s') \longrightarrow always\_inv(s', A'_1)$$

.

2. After substitution of extra invariants satisfying the derived pattern L_4 has the form:

$$P1inv(s, t, t_1, A_1, A'_2, A'_3) \longrightarrow consecutive(s, s') \longrightarrow (\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'(s, s_1) \longrightarrow A'(s', s_1)) \wedge A'(s', s') \longrightarrow P1inv(s', t, t_1, A_1, A'_2, A'_3)$$

   where the parameter A_1 in the lemma is unprimed because its value cannot contain nested patterns and therefore its value in an extra invariant coincides with the value of the corresponding parameter in the requirement. Here the parameter A'_1 in the lemma L_4 from the previous step is renamed to A' to avoid name collisions.
3. the formula C has the form: $(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'(s, s_1) \longrightarrow A'(s', s_1)) \wedge A'(s', s')$
4. After substitution of the values of the parameters A' of the outer pattern `always_inv` C has the form: $(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge (\neg A_1(s_1) \vee constrained\_until\_inv(s, s_1, t, t_1, A'_2, A'_3)) \longrightarrow (\neg A_1(s_1) \vee constrained\_until\_inv(s', s_1, t, t_1, A'_2, A'_3)) \wedge (\neg A_1(s') \vee constrained\_until\_inv(s', s', t, t_1, A'_2, A'_3))$
5. Additional conjuncts are not added to C because there are no nested past-time patterns.
6. The formula C contains nested patterns. Therefore the loop for transforming the lemma is performed in which the transformation rules are applied:
   (a) At the first iteration, the rule 6a for a conjunction under universal quantifier is applied, and after the iteration, c has the form: $((\forall s_1.e(s_1) \wedge s_1 \leq s \wedge \neg A_1(s_1) \longrightarrow \neg A_1(s_1)) \wedge (\forall s_1.e(s_1) \wedge s_1 \leq s \wedge constrained\_until\_inv(s, s_1, t, t_1, A'_2, A'_3) \longrightarrow constrained\_until\_inv(s', s_1, t, t_1, A'_2, A'_3))) \wedge (\neg A_1(s') \vee constrained\_until\_inv(s', s', t, t_1, A'_2, A'_3))$
   (b) At the second iteration, the rule 6b for a future-time pattern under universal quantifier is applied, and after the iteration, c has the form: $((\forall s_1.e(s_1) \wedge s_1 \leq s \wedge \neg A_1(s_1) \longrightarrow \neg A_1(s_1)) \wedge (\forall s_1.e(s_1) \wedge s_1 \leq s \wedge$

$A'_2(s,s_1) \longrightarrow A'_2(s',s_1)) \wedge (forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_3(s,s_1) \longrightarrow A'_3(s',s_1)) \wedge (t_1(s) = 0 \vee A'_3(s',s') \wedge t_1(s) \leq t \vee A'_2(s',s') \wedge t_1(s) < t_1(s'))) \wedge (\neg A_1(s') \vee constrained\_until\_inv(s',s',t,t'_1,A'_2,A'_3))$

(c) At the third iteration, the identity rule for a future-time pattern 6c is applied, and after the iteration, c has the form: $((\forall s_1.e(s_1) \wedge s_1 \leq s \wedge \neg A_1(s_1) \longrightarrow \neg A_1(s_1)) \wedge (\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_2(s,s_1) \longrightarrow A'_2(s',s_1)) \wedge (forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_3(s,s_1) \longrightarrow A'_3(s',s_1)) \wedge (t_1(s) = 0 \vee A'_3(s',s') \wedge t_1(s) \leq t \vee A'_2(s',s') \wedge t_1(s) < t_1(s'))) \wedge (\neg A_1(s') \vee (A'_3(s',s') \vee A'_2(s',s') \wedge t_1(s') > 0))$

7. Now the formula C does not contain nested patterns, and the following lemma L_4 is returned as lemma for the pattern $P1$ satisfying the lemma pattern LP_4:

$$
\begin{array}{l}
P1inv(s,t,t1,A1,A2',A3') \longrightarrow \\
consecutive(s,s') \longrightarrow \\
((\forall s_1.e(s_1) \wedge s_1 \leq s \wedge \neg A_1(s_1) \longrightarrow \neg A_1(s_1)) \wedge \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_2(s,s_1) \longrightarrow A'_2(s',s_1)) \wedge \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A'_3(s,s_1) \longrightarrow A'_3(s',s_1)) \wedge \\
(t1(s) = 0 \vee A2'(s',s') \wedge t1(s \leq t \vee A2'(s',s')t1(s) < t1(s'))) \wedge \\
(\neg A1(s') \vee A3'(s',s') \vee A2'(s',s') \wedge t1(s') > 0) \longrightarrow \\
P1inv(s',t,t1,A1,A2',A3')
\end{array}
$$

Let's describe constructing the lemma L_5 for this derived requirement pattern:

1. Take the lemma L_5 for the outer pattern `always`. It is following:

$$
\begin{array}{l}
e(s) \longrightarrow \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A(s_1) \longrightarrow A'(s_1)) \wedge \\
always(s,A) \Longrightarrow \\
always(s,A')
\end{array}
$$

2. After substitution of the extra invariant and requirement pattern instances L_5 has the form:

$$
\begin{array}{l}
e(s) \longrightarrow \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A(s_1) \longrightarrow A'(s_1)) \wedge \\
P1inv(s,t,t_1,A_1,A_2,A_3) \Longrightarrow \\
P1(s,t,A_1,A'_2,A'_3)
\end{array}
$$

   where the parameter A_1 in the conclusion of the lemma is unprimed because its value cannot contain nested patterns.
3. The formula C has the form: $(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A(s_1) \longrightarrow A'(s_1))$
4. After substitution of the values of the parameters A and A' of the outer pattern `always` C has the form: $(forall s_1.e(s_1) \wedge s_1 \leq s(\neg A_1(s_1) \vee constrained\_until\_inv(s_1,s,t,t_1,A_2,A_3)) \longrightarrow (\neg A_1(s_1) \vee constrained\_until(s_1,s,t,A'_2,A'_3)))$
5. The formula C contains nested patterns. Therefore the loop is performed:

(a) At the first iteration, the rule 6a is applied, and after the iteration, c has the form: $(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge \neg A_1(s_1) \longrightarrow \neg A_1(s_1)) \wedge (\forall s_1.e(s_1) \wedge s_1 \leq s \wedge constrained\_until\_inv(s_1, s, t, t_1, A_2, A_3) \longrightarrow constrained\_until(s_1, s, t, A'_2, A'_3))$

(b) At the second iteration, the rule 6b is applied, and after the iteration, c has the form: $(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge \neg A_1(s_1) \longrightarrow \neg A_1(s_1)) \wedge (\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A_2(s_1) \longrightarrow A'_2(s_1)) \wedge (forall s_1.e(s_1) \wedge s_1 \leq s \wedge A_3(s_1) \longrightarrow A'_3(s_1)) \wedge t_1 \leq t$

6. Now the formula C does not contain nested patterns, and the following lemma L_4 is returned:

$$
\begin{array}{l}
P1inv(s, t, t_1, A_1, A_2, A_3) \Longrightarrow \\
e(s) \longrightarrow \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge \neg A_1(s_1) \longrightarrow \neg A_1(s_1)) \wedge \\
(\forall s_1.e(s_1) \wedge s_1 \leq s \wedge A_2(s_1) \longrightarrow A'_2(s_1)) \wedge \\
(forall s_1.e(s_1) \wedge s_1 \leq s \wedge A_3(s_1) \longrightarrow A'_3(s_1)) \wedge \\
t_1 \leq t \Longrightarrow \\
P1(s, t, A_1, A'_2, A'_3)
\end{array}
$$

After identifying A_2 with A'_2 and A_3 with A'_3 and excluding identically true premises in the lemmas L_4 and L_5 we obtain the following lemmas L_6 and L_7:

$$
\begin{array}{l}
L_6: \\
P1inv(s, t, t_1, A_1, A_2, A_3) \longrightarrow \\
consecutive(s, s') \longrightarrow \\
(t_1 = 0 \vee A_3(s') \wedge t_1 \leq t \vee A_2(s') \wedge t_1 < t'_1) \wedge \\
(\neg A_1(s') \vee (A_3(s') \vee A_2(s') \wedge t'_1 > 0)) \\
P1inv(s', t, t'_1, A_1, A_2, A_3)
\end{array}
$$

$$
\begin{array}{l}
L_7 \\
P1inv(s, t, t_1, A_1, A_2, A_3) \Longrightarrow \\
e(s) \Longrightarrow \\
t_1 \leq t \Longrightarrow \\
P1(s, t, A_1, A_2, A_3)
\end{array}
$$

In the requirement for the hand dryer control program,

$$
\begin{array}{l}
A_1(s_1) \equiv s_1[hands] = False; \\
A_2(s_2) \equiv s_2[dryer] = True \wedge s_2[hands] = False; \\
A_3(s_3) \equiv s_3[dryer] = False \vee s_3[hands] = True.
\end{array}
$$

We also need to determine the value of the parameter t_1 in the extra invariant pattern to define an extra invariant for this requirement. Currently we do not have an efficient algorithm for finding the values of extra invariant patterns. Therefore, in this work, we specify the parameter values manually. In the extra invariant for the considered requirement,

$$t_1 \equiv \texttt{if}\ \ getPstate(s, Ctrl) = drying\ \ \texttt{then}\ \ ltime(s, Ctrl)\ \ \texttt{else}\ \ 0.$$

7 Related Work

7.1 Loop Invariant Generation

There is a wide variety of methods of finding loop invariants. These include abstract interpretation [9], induction-iteration method [26], template-based methods [7], recurrence analysis [16], using failed proof attempts [25] and invariant strengthening on demand [17], dynamic analysis [22] and machine learning [23]. Abstract interpretation and template-based methods are the most common approaches to the static loop invariant inference [10]. Let's consider the works closest to this one on the automatic generation of loop invariants.

Template-based methods are most successfully applied within the domain of linear arithmetic [3]. In these methods, a template is specified that defines the general form of invariants and has parameters. Then constraints on the values of the parameters are found based on the program and the requirement being verified, and these constraints are solved to obtain the values of the parameters. In [7], a method of linear loop invariants generation based on constraint solving. Invariants have he form of linear inequalities. Authors generate constraints on the template parameters that are coefficients in the inequality. These constraints ensure that the invariant is true 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 constraint can be solved by quantifier elimination. But because quantifier elimination is a costly process, 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 (hidden in patterns). Currently specify the values of pattern parameters 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.

In [14], a slightly different template-based approach is used. Loop invariant templates are quantifier-free formulas containing program variables and parameters. As an example, the author considers a template in the form of a polynomial equation. To obtain constraints on template parameters, the template is substituted into verification conditions instead of the loop invariant. The obtained constraints contain universal quantifiers over program variables. Quantifier elimination can be used to obtain constraints containing only parameters. Simplification of a constraint corresponding to one path in the loop can allow one to find the values of certain parameters and simplify the loop invariant template that can be substituted into the verification condition for another path. Heuristics can be used to determine the order of processing paths. Unlike to that work, our extra invariant patterns contain quantifiers. Similar to that work, we can substitute patterns into verification conditions, but it lead to constraints with quantifiers over update states. Such constraints can be reduced to quantifier-free formulas using our lemmas used for proving verification conditions.

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 finding such expressions to to finding constants, we could consider instances of constraints for different paths in the program.

The paper [27] proposes an approach in which SMT solving is combined with machine learning to generate loop invariants. A general template in which not only constants are unknown is set. Then the machine learning model suggested by the authors prunes the general template to a sub-template the parameters of which are only constants. Next, template solving is performed. At this stage, an SMT solver is used to find the values of these constants based on a set of counterexamples. After finding the values of the constants an SMT solver is used to check that the obtained invariant allows proving verification conditions. If a counterexample is found for some verification condition, this counterexample is added to the counterexample set. Found counterexamples are used at the next iterations for template solving and reinforcement learning. If the template solving fails, the machine learning model tries to find another sub-template. Our extra invariants patterns can be considered as general templates because the pattern parameters are not constants, but expressions and their general form is determined. A sun-template in this case will be an extra invariant pattern in which the values of boolean parameters and conditions in conditional expressions are substituted. We could try to use machine learning for finding such a sub-template, but also plan to develop a heuristic algorithm for this purpose. Unlike to that work, we do not plan to use counterexamples for extra invariant generation. Another difference is that we do not use SMT solvers for proving verification conditions. We use automated proof methods of Isabelle/HOL, but their application is preceded by the application of our lemmas.

Template-based methods may suffer from the problem of template size growth [15]. In that work, the authors use polynomial invariant templates. The problem is that the polynomial templates of high degrees contain a large number of monomials. This makes the generation of high degree invariants less scalable. To make template-based methods more efficient, the authors define generalized homogeneous polynomials as polynomials satisfying the quantity dimension principle. The desired invariant is represented as a sum generalized homogeneous polynomials. In our work, the problem of the template size may be related to the large number of branches in conditional expressions in the pattern parameters values. We need to develop heuristics to solve this problem.

Let's consider some related works in which other methods of invariant generation are used. In STeP [18], two approaches to invariant generation are used:

the bottom-up approach in which in variants 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. 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. This paper is devoted to top-down 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.

Another method of loop invariant generation is recurrence analysis [16]. In recurrence analysis, the loop body for which invariant is generated is represented as a set of recurrence equations. Then the recurrence equations are solved to obtain a non-recursive function of the initial variable values and the loop counter. Next, the loop counter is eliminated, and identities among the program variables are obtained as loop invariants. In our work, recurrences appears in constraints on the extra invariants patterns parameters. But it is a particular case of recurrences, and we have patterns for their solutions. Therefore we do not need to solve recurrence equations.

7.2 Deductive Verification of temporal requirements

In some works, deductive verification of reactive systems with temporal requirements is performed by reducing the verification to proving first-order formulas. In [19], an approach based on temporal requirements classification is proposed. The temporal formula describing the requirement to be proven is transformed into the canonical form of temporal formulas in which a set of future modalities is applied to past formulas, and the program is represented as a transition system. Then the suitable proof rule is applied depending on the class of the formula determined by its type in the canonical form that reduces the temporal formula to non-temporal assertions. The author presented proof rules for invariance, response and reactivity properties. For example, the rule for invariance reduces an invariance requirement to three premises: 1) the initial condition implies an auxiliary assertion; 2) the auxiliary assertion preserves after each transition; 3) the auxiliary assertion implies the property being proven. Since the temporal requirements are specified as control loop invariants in our work, the rule for proving invariance is used. However extra invariants that are the auxiliary assertions are not non-temporal assertions, but formulas relating the values of variables at different points in time. We define lemmas allowing reducing such invariants to quantifier-free first-order formulas relating variable values in one point. Our approach is based on requirement patterns. Unlike that work, we do not transform temporal properties into the canonical form, but define lemmas

for each derived requirement pattern by induction on the construction of the derived pattern.

In [11], a method for deductive verification of reactive systems is introduced in which requirements are specified using temporal logic CTL* [6]. A set of proof rules is presented for deductive verification, and it is demonstrated that these rules are sound and relatively complete. This method employs two transformations The first transformation decomposes a CTL* formula into basic state formulas— formulas that do not contain embedded path quantifiers, and the second transformation decomposes a general path formula into basic path formulas—formulas lacking temporal operators other than the main one. Since we do not verify branching-time temporal properties in our work, let's focus on the second decomposition. Each basic path formula is substituted with a new boolean variable. This method is grounded in the concept of temporal testers. A temporal tester can be viewed as a transition system where the input consists of variables included in the path formula, and the output is a new variable that is true iff the corresponding temporal formula holds at the same position. That paper discusses the testers for basic path formulas and the incremental construction of testers for general path formulas. These testers are then executed in parallel with the verified system. It is noted that this approach is better than the one that was used in [19, **?**] beauce it does not require to bring a formula to the canonical form. We do not use temporal testers, but similar to the construction of the testers, we introduce basic patterns and lemmas for them and incrementally construct derived extra invariant patterns and lemmas for derived requirement patterns. In addition, as in that work, we consider formulas about both the future and the past, although in our case the future is bounded. But neither that work nor the previously considered work addresses quantitative temporal requirements.

Another approach is to reduce proving that a program satisfies temporal requirements to proving temporal formulas. In [2], the authors discuss deductive verification of robotic behaviors. Temporal logic is used to specify temporal requirements. A transition system describing the robotic behavior is represented in the form of a set of LTL formulas, and additional constraints are specified. Then the resolution-based theorem prover for LTL TRP++ [13] is used to prove that conjunction of the formulal representing the transition system and the additional constraints implies the desired property. Unlike that work, we use rules of axiomatic semantics, similar to the rules of Hoare logic, and generate first-order verification conditions instead of describing program behavior by formulas of a temporal logic. Also, in that article, the authors do not model low-level control, but only high-level behavior.

8 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 basic set of patterns used in require-

ments are defined. For each such basic pattern, the corresponding basic pattern used in extra invariants and lemmas are defined. These lemmas reduce proving formulas containing instances of these patterns to proving more simple formulas that do not contain these instances. Then the basic patterns used in requirements can be combined using conjunction, disjunction and nesting to define derived requirement patterns. For each derived requirement pattern, the corresponding extra invariant pattern and lemmas for proving verification conditions for requirements and extra invariants satisfying these patterns are constructed. In this paper, we have described how to construct the extra invariant patter corresponding to a derived requirement pattern. We have also developed algorithms for generation of lemmas for derived patterns and scripts for proving these lemmas in Isabelle/HOL proof assistant.

The approach proposed in this paper will allow one to automatically determine the extra invariant pattern lemmasneeded 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.

In the future, we plan to develop tools for generation of the extra invariant pattern, 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 invariants patterns. This could allow us to automate deductive verification of some class of process-oriented programs in poST language. Another possible direction of the futuyre work is extending our approach to new classes of poST programs, in particular, programs with arithmetic operations and loops.

References

1. Anureev, I., Garanina, N., Liakh, T., Rozov, A., Zyubin, V., Gorlatch, S.: Two-step deductive verification of control software using reflex. In: Bjørner, N., Virbitskaite, I., Voronkov, A. (eds.) Perspectives of System Informatics. pp. 50–63. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-37487-7$_5$
2. Behdenna, A., Dixon, C., Fisher, M.: Deductive verification of simple foraging robotic behaviours. International Journal of Intelligent Computing and Cybernetics **2**(4), 604–643 (2009)
3. Breck, J., Cyphert, J., Kincaid, Z., Reps, T.: Templates and recurrences: better together. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 688–702 (2020)
4. Chernenko, I., Anureev, I.S., Garanina, N.O., Staroletov, S.M.: A temporal requirements language for deductive verification of process-oriented programs. In: 2022 IEEE 23rd International Conference of Young Professionals in Electron Devices and Materials (EDM). pp. 657–662. IEEE (2022)
5. Chernenko, I.M., Anureev, I.S.: Requirement-dependent extra invariant patterns in deductive verification of post programs. In: 2024 IEEE 25th International Conference

of Young Professionals in Electron Devices and Materials (EDM). pp. 1900–1905 (2024). https://doi.org/10.1109/EDM61683.2024.10615007
6. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R., et al.: Handbook of model checking, vol. 10. Springer (2018)
7. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15. pp. 420–432. Springer (2003)
8. Commission, I.E., et al.: Programmable controllers-part 3: Programming languages. IEC 61131-3 (Ed. 2.0) (2002)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 238–252 (1977)
10. Furia, C.A., Meyer, B., Velder, S.: Loop invariants: Analysis, classification, and examples. ACM Computing Surveys (CSUR) **46**(3), 1–51 (2014)
11. Gabbay, D.M., Pnueli, A.: A sound and complete deductive system for ctl* verification. Logic Journal of the IGPL **16**(6), 499–536 (2008)
12. 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 pp. 345–373 (2019)
13. Hustadt, U., Konev, B.: Trp++ 2.0: A temporal resolution prover. In: International Conference on Automated Deduction. pp. 274–278. Springer (2003)
14. Kapur, D.: Automatically generating loop invariants using quantifier elimination. Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2006)
15. Kojima, K., Kinoshita, M., Suenaga, K.: Generalized homogeneous polynomials for efficient template-based nonlinear invariant synthesis. In: International Static Analysis Symposium. pp. 278–299. Springer (2016)
16. Kovács, L.: Reasoning algebraically about p-solvable loops. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 249–264. Springer (2008)
17. Leino, K.R.M., Logozzo, F.: Loop invariants on demand. In: Asian symposium on programming languages and systems. pp. 119–134. Springer (2005)
18. Manna, Z., Bjørner, N., Browne, A., Chang, E., Colón, M., de Alfaro, L., Devarajan, H., Kapur, A., Lee, J., Sipma, H., et al.: Step: The stanford temporal prover. In: TAPSOFT'95: Theory and Practice of Software Development: 6th International Joint Conference CAAP/FASE Aarhus, Denmark, May 22–26, 1995 Proceedings 20. pp. 793–794. Springer (1995)
19. Manna, Z., Pnueli, A.: Completing the temporal picture. Theoretical Computer Science **83**(1), 97–130 (1991)
20. Matichuk, D., Murray, T., Wenzel, M.: Eisbach: A proof method language for isabelle. J. Autom. Reason. **56**(3), 261–282 (mar 2016). https://doi.org/10.1007/s10817-015-9360-2, https://doi.org/10.1007/s10817-015-9360-2
21. Paulson, L.C., Nipkow, T., Wenzel, M.: From lcf to isabelle/hol. Formal Aspects of Computing **31**, 675–698 (2019)
22. Perkins, J.H., Ernst, M.D.: Efficient incremental algorithms for dynamic detection of likely invariants. In: proceedings of the 12th ACM SIGSOFT twelfth International Symposium on Foundations of Software Engineering. pp. 23–32 (2004)
23. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. Advances in Neural Information Processing Systems **31** (2018)

24. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verification and program synthesis. International Journal on Software Tools for Technology Transfer **15**, 497–518 (2013)
25. Stark, J., Ireland, A.: Invariant discovery via failed proof attempts. In: International Workshop on Logic Programming Synthesis and Transformation. pp. 271–288. Springer (1998)
26. Suzuki, N., Ishihata, K.: Implementation of an array bound checker. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 132–143 (1977)
27. Yu, S., Wang, T., Wang, J.: Loop invariant inference through smt solving enhanced reinforcement learning. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 175–187 (2023)
28. Zyubin, V.E.: Hyper-automaton: A model of control algorithms. In: 2007 Siberian Conference on Control and Communications. pp. 51–57. IEEE (2007). https://doi.org/10.1109/SIBCON.2007.371297