УДК 004.415.52+519.681

Enhancing Verification Condition Generation for Reflex Programs Through Simple Static Analysis*

Ishchenko A.D. (Institute of automatics and electrometry, Siberian Branch of the Russian Academy of Sciences)

Control systems play a crucial role in various domains necessitating high reliability and proof of correct software operation. To address this, formal methods, such as deductive verification, are employed to ensure the correctness of safety-critical software mathematically. This process involves generating verification conditions from the program's axiomatic semantics and its requirements. Main disadvantage of manual verification condition generation is its labor-intensity and prone to human error, which leads to the development of automated verification condition generators. However, they produce excessive or overly complex verification conditions that do not accurately reflect the possible program's operational paths. To mitigate the second issue, domain-oriented languages like Reflex have been introduced, which adopt a process-oriented paradigm to streamline programming and simplify verification condition generation. Nevertheless, the inherent switch-case structure of Reflex can lead to an increase in the number of verification conditions, exacerbating the first issue and complicating the selection of relevant ones. This paper proposes a simple static analysis system designed to optimize the generation of verification conditions for Reflex programs, enhancing the efficiency of the verification process. It is based on attaching attributes to different statements. Then, throughout the verification condition generation these attributes are collected and checked to be compatible with previously collected. If attributes are incompatible then verification condition are discarded.

Key words: process-oriented, static analysis, attributes, deductive verification

1. Introduction

Control systems are widely used in many areas: from the Internet of Things to programmable industrial controllers. This requires advanced reliability for such systems and, in particular, proof of correct operation of used software.

Nowadays, the most common method to ensure correctness is to test the system on digital models or real-world control objects. It is popular for its simplicity and relative cheapness. However, this method is incomplete because it does not cover all possible cases. This leads to

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

a danger of appearing of rare errors among widely used devices which looked completely safe.

To ensure the correctness of safety-critical software, formal methods are used. They use formal semantics of the program and its requirements to simplify this challenge by transforming it into some mathematical problem.

One of these formal methods is deductive verification [5]. It requires defining the axiomatic semantics of programming language in some logical inference system and formalizing requirements. Then program and its requirements are transformed into a set of logical formulas. This process is called verification condition (VC) generation. Doing it manually is time-consuming and dangerous because of human factors. To automate this process, verification condition generators are developed. A generator goes through all possible paths of control flow and creates VCs for them.

However, the naive realization of the generator may create a lot of excess verification conditions that do not represent the ways the program really could work. Therefore, it is important to find ways for reducing the amount of VCs and their simplification.

For simplification of creating and verification of control software special domain-oriented languages are used. One of them is Reflex language [10] created for writing control programs. It is designed in the style of a process-oriented paradigm. This paradigm represents the program as a list of processes that are executed sequentially once in a certain period called activation time thus forming a control loop. Each process consists of several active states created by the programmer and two inactive states: stop and error. Each state is a list of instructions that include both ordinary imperative statements (for example, conditional statements) and special process-oriented statements designed to interact with other processes. The use of the Reflex language rebuilds the programming style for control programs, which leads, in particular, to the definition of VCs in terms of control processes and their states and thus makes them conceptually simpler. However, this also leads to an increase in the number of VCs caused by the switch-case nature of processes of a Reflex program, and some of these VCs are 'false-positive' since they correspond to the paths of the program that are non-reachable in its actual execution.

In this paper, we propose a static analysis system for Reflex programs that finds such VCs to optimize the process of VCs generation.

2. Reflex language

A Reflex program consists of processes, process states, process-oriented statements and C-like statements.

For interaction with processes and their states, the following instructions are defined:

- *restart*, *start* p – sets the current process or process p into its first state;
- *stop*, *stop* p – sets current process or process p into stop state;
- *error*, *error* p – sets the current process or process p into error state;
- *set state* s, *set next state* – sets the current process into state s or state defined after the current one;
- *reset timer* – set local time of the current process to 0;
- *process* p *in state* k – checks whether process p is in state kind $k \in \{$*active*, *inactive*,*stop*,*error*$\}$.

Besides them, the following C-like constructs are used:

- C-like expressions;
- if-else statements;
- switch-case statements.

3. Static analysis

An increase in the number of VCs occurs in processing the following statements:

- *process ...* – enumeration of different process states;
- *timeout t ...* – checking cases when timeout t exceeded and when not;
- *if (expr) ... else ...* – checking cases when `expr` is true or false;
- *switch (expr) {case ...}* – checking when `expr` corresponding to different cases;

In this paper we focus on the first two cases, and for the second one only analyze cases when timeout parameter t is presented by a constant value. For example (Fig. 1), if in the process of VC generation some process A starts another process B that contextually appears after the first one, then, when process B will be processed, it must be in its first state.

Before the analysis program is transformed into canonical form with the conversion of short forms of operators into full ones:

- *restart* $\longrightarrow$ *start* **p** where **p** is the current process;
- *stop* $\longrightarrow$ *stop* **p** where **p** is the current process;
- *error* $\longrightarrow$ *error* **p** where **p** is the current process;
- *set next state* $\longrightarrow$ *set state* **s** where **s** is the next state of the current process.

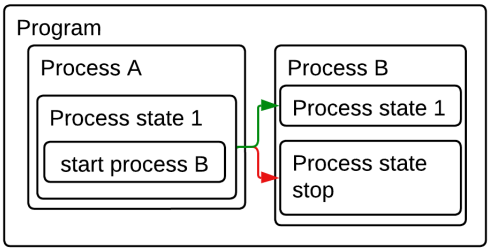

Figure 1: Possible and impossible paths. They are shown by green and red arrows correspondingly

The result of static analysis is a set of attributes. They are used in verification condition generation to avoid the creation of VCs for impossible paths. They are defined for declarations of processes, process states and statements included in the bodies of process state declarations. Let **p**.A means the value of the attribute A of the declaration of process **p**, **p**.**s**.A does the value of attribute A of declaration of the state **s** of process **p**, **p**.**s**.**st**.A does the value of attribute A after execution of the statement **st** of state **s** of process **p**.

The following list describes evaluated attributes:

1. $p.state := s$ – contains the current state of process p;
2. $p.reachE := true/false$ – shows whether process p ever reaches error state;
3. $p.reachS := true/false$ – shows whether process p ever reaches stop state;
4. $p.startS := true/false$ – shows whether process p could start in stop state;
5. $p.s.st.procChange := f$ – contains the partial function from process to $\{start, stop\ or\ error\}$ where, for example, f(p')=start means that process p' was started after execution of statement p of process state p of process p;
6. $p.s.st.reset := true/false$ – shows whether timer was reset;

Attributes $reachE$, $reach$ and $reset$ have $false$ value by default. Attribute $startS$ has default value $false$ for the first process and $true$ for other processes. Attributes $state$ and $procChange$ are not defined initially.

A general static analysis scheme is presented in figure 2. The algorithm takes an abstract syntax tree (AST) of the analyzed program and sets initial attribute values to corresponding statements. Then, the lifting algorithm is applied to evaluate attributes of more general constructions. After that, attributes of more complex properties are set. In conclusion, AST with set attributes and attributes compatibility rules are provided to the VC generation algorithm.

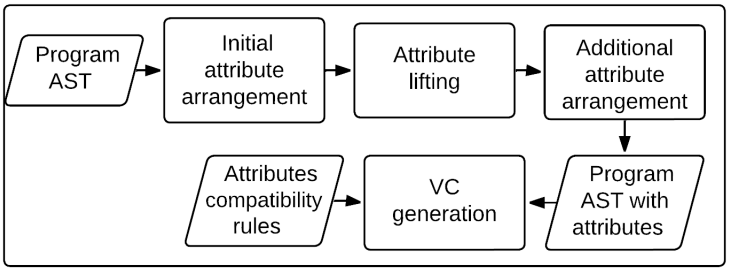

Figure 2: Static analysis scheme

3.1. Initial attribute arrangement

Firstly, we define attributes *start*, *stop*, *error*, *reset*. To do this we traverse AST of a program and set attributes values in accordance on type of statement *st*:

```
if(st ≡ start p′) then
    st.procChange.add(p′,start)
    if (p ≡ p′) then st.reset := true
if(st ≡ stop p′) then
    st.procChange.add(p′,stop)
    if (p′≡p) then st.reset := true
if(st ≡ error p′) then
    st.procChange.add(p′,stop)
    if (p′≡p) then st.reset := true
if(st ≡ reset timer) then st.reset := true
if(st ≡ set state s′) then st.reset := true
```

where s and p are current state and process, respectively, and $\equiv$ denotes syntactic coincidence.

Attributes *reachE* and *reachS* define whether a process ever reaches states *stop* and *error* by being put into them by some processes. So initialization of these attributes is done by the following algorithm:

```
if(∃p′∈processes(r)|∃s∈states(p′)|(error p∈body(s))
    then p.reachE := true)
if(∃p′∈processes(r)|∃s∈states(p′)|(stop p∈body(s))
    then p.reachS := true)
```

where r is a program, processes(r) returns list of processes of r, states(p) returns list of states of process p and body(s) returns a list of all statements of state s.

3.2. Lifting algorithm

Lifting is an algorithm designed for evaluating more general properties by lifting attributes to higher-level statements. Attributes may be lifted to them in two cases: if the attribute belongs

to a statement that is part of a linear sequence of statements, or if the attribute appears at all branches of a statement with several paths: *if-else*, *switch-case*.

For a statement sequence *st* lifting is done by the following algorithm:

```
f := ⊥; reset := false;
for st′ in statements(st)
    if (st'.reset) then reset := true;
    f := f ∪ st′.procChange;
st.reset := reset; st.procChange := f;
```

where $statements(st)$ returns a sequence of statements of compound statement *st*.

For a branching statement *st*, the algorithm has the following form:

```
reset := true; procChange := (first(statements(st))).procChange;
for st′ in statements(st)
    if (!st′.reset) then reset := false;
    procChange := procChange ∩ st′.procChange;
st.reset := reset; st.procChange := procChange;
```

where $first(l)$ returns the first element of the list l.

3.3. Additional attribute arrangement

The definition of the attribute *startS* depends on the value of this attribute in previous processes and the process starting in them. It is done by the following algorithm:

```
for p in processes(r)
    if (∃ p′ ∈ processes(r)| p′.id < p.id ∧ p′.startS = false ∧
        p ∈ first(states(p′)).start) then p.startS := false;
```

where *id* is the field defining the number of the process in the order of definition in the program. In the following example (Fig. 3) processes **P2** and **P4** become with attributes *startS* equal to false because these processes will be start in the first iteration by process **P1**. Process **P3** remains with *startS* equal to true because it is executed before **P4**, so statement $\{startP3\}$ will take effect only on the second iteration.

The value of attribute *state* is defined during the generation of VCs. If the generation algorithm reaches the state s of process p, the attribute $p.state$ is set into the s.

3.4. VC generation

After all attributes are set and lifted to the corresponding structure verification condition generation starts. The original VC generation algorithm is described in [6] and realized on GitHub [1]. It traverses the abstract syntax tree building up the list of preconditions *acc* and

[1] https://github.com/bearhug15/ReflexVCG/tree/before_analysis_add

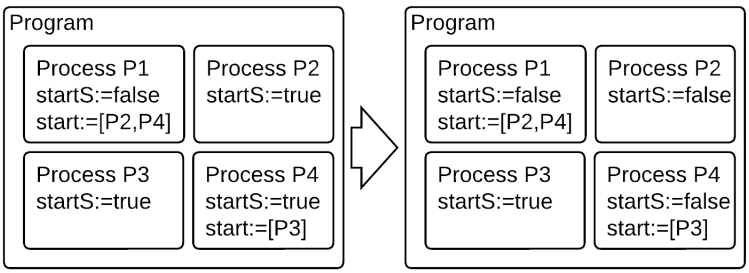

Figure 3: Attributes change due to start attribute

stack of marks made on statements which. Later when the formation of the current VC ends it jumps to the next mark, drops excess preconditions, and continues VC generation from the new place. Usage of attributes updates described algorithm. In addition to the *acc*, a *passed* list is filled. *Passed* describes the passed path as a sequence of language constructs which allows excluding some impossible continuations of this path, based on checking the compatibility of values of attributes of the constructs. After the completion of VC generation and dropping of excess preconditions excess path also dropped.

Updated VC generation function for statement *st* of state *st* of process *p* with checking of VC possibility and filling of *passed* list of statements is done by the following algorithm:

```
if (checkComp(passed, p,s,st)) then
    add(passed, st); updatePrec(st, acc);
else cutToMark(passed, acc);
```

where *add* adds structure *st* into *passed* list, *checkComp* checks whether attributes of *st* are compatible with attributes of structures in *passed* path and return true if they are compatible, *cutToMark* cuts *passed* and *acc* lists to last mark. *updatePrec* updates generated VC based on the existing generation algorithm. The current realization of *checkComp* is done in table 1:

Realization of created algorithms is on Github [2].

4. Related works

The increasing complexity of programs has led to a significant rise in the number of verification conditions (VCs), posing a major challenge in deductive program verification. As highlighted by Hähnle et al. [4] verification of cyber-physical systems, including programmable logic controllers, is itself one of the modern challenges in the field of deductive verification, and this issue only makes it harder. Additionally, various techniques aimed at simplifying VCs

[2]https://github.com/bearhug15/ReflexVCG

Condition	Return value
$p.reachE = false \wedge p.state = error$	$false$
$p.reachS = false \wedge p.startS = false \wedge p.state = stop$	$false$
$\exists p', s', st' \in passed.\ p \in p'.s'.st'.start \wedge p.state \neq first(p)$	$false$
$\exists p', s', st' \in passed.\ p \in p'.s'.st'.stop \wedge p.state \neq stop$	$false$
$\exists p', s', st' \in passed.\ p \in p'.s'.st'.error \wedge p.state \neq error$	$false$
$\exists st' \in passed.\ p'.s'.st'.reset = true \wedge st \equiv \{timeout \ldots\}$	$false$
Otherwise	$true$

Table 1: Table of *checkComp* rules.

can inadvertently exacerbate this problem. For instance, Leino et al. [7] propose a method that involves splitting a verification condition into multiple separate conditions, where the conjunction of these conditions remains equivalent to the original. While such techniques may offer potential benefits for program verification, they can complicate the situation for Reflex programs. This complexity arises from the finite state machine abstraction inherent in the Reflex language, which generates a control flow graph characterized by high coupling and low cohesion. Consequently, splitting a single condition can lead to the fragmentation of numerous conditional paths, necessitating the automatic discarding of irrelevant conditions at generation time, which relies on various forms of static analysis.

Couchot [2] introduces a graph-based technique for reducing verification conditions, utilizing two types of graphs: the constant dependence graph and the predicate dependence graph. These graphs facilitate the checking of VC satisfiability and the discharge of unsatisfiable conditions. However, this approach has notable drawbacks, including the requirement to construct a complete VC prior to checking and its reliance on SAT solvers. Such dependencies can significantly extend the VC generation process, making program development less efficient. In contrast, our approach focuses solely on syntax analysis, which is simpler and more time-efficient.

Another perspective on the challenges we address involves identifying unreachable states. In this context, a combination of process state values and timeout invocations is treated as a single state, with transitions between these combinations representing state transitions through iterations of the control loop. Several methods have been proposed to tackle this issue [1, 9]. They are based on three steps: 1)reachable state space over-approximation; 2) debugging; and 3) spurious solution detection. Changing in the last step allows us to vary the accuracy of

methods to discard more unreachable states. The advantage of these methods is that they may consider discrete time, which in our case is equivalent to loop iterations. However, these methods require explicit state formulations, which can be memory-intensive and rely on slow model-checking techniques. Modification for improvement in analysis accuracy also significantly slows it.

Additionally, we can frame our problem as a context-free language (CFL) reachability problem and explore reachability analysis techniques [3, 8]. Given that the reachability of each process state may be constrained by various statements from earlier parts of the program, this approach resembles a variant of the all-pairs S-path problem. While applicable in scenarios where unnecessary processing of *stop* and *error* states can be eliminated, this method lacks completeness, as it does not adequately account for more complex program behaviors.

5. Conclusion

This paper presents our approach to reducing the amount of verification conditions corresponding to the unreachable paths. We created a set of attributes associated with path elements and a set of heuristics that detect incompatible combinations of attribute values on the paths.

In the future, we plan to expand this analysis with more attributes and heuristics to consider more unreachable paths.

References

1. Berryhill Ryan, Veneris Andreas G. Methodologies for Diagnosis of Unreachable States via Property Directed Reachability // IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. — 2018. — Vol. 37. — P. 1298–1311. — URL: https://api.semanticscholar.org/CorpusID:13169698.
2. Couchot Jean-François, Giorgetti Alain, Stouls Nicolas. Graph Based Reduction of Program Verification Conditions // arXiv preprint arXiv:0907.1357. — 2009.
3. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis / Qirun Zhang, Michael R Lyu, Hao Yuan, Zhendong Su // Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. — 2013. — P. 435–446.
4. Hähnle Reiner, Huisman Marieke. 24 challenges in deductive software verification // 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements / EasyChair. — 2017. — P. 37–41.

5. Hähnle Reiner, Huisman Marieke. 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. — URL: https://doi.org/10.1007/978-3-319-91908-9_18.
6. Ishchenko Artyom D., Anureev Igor S. Verification Condition Generator for Process-Oriented Programs in Reflex Language Using Isabelle/HOL // 2024 IEEE 25th International Conference of Young Professionals in Electron Devices and Materials (EDM). — 2024. — P. 1820–1825.
7. Leino K Rustan M, Moskal Michał, Schulte Wolfram. Verification condition splitting // Submitted manuscript, September. — 2008.
8. Reps Thomas. Program analysis via graph reachability // Information and software technology. — 1998. — Vol. 40, no. 11-12. — P. 701–726.
9. Suda Martin. Property directed reachability for automated planning // Journal of Artificial Intelligence Research. — 2014. — Vol. 50. — P. 265–319.
10. Zyubin Vladimir Evgenievich, Liakh Tatiana Viktorovna, Rozov Andrei Sergeevich. Reflex language: a practical notation for cyber-physical systems // System Informatics. — 2018. — no. 12. — P. 85–104. — URL: https://doi.org/10.31144/si.2307-6410.2018.n12.p85-104.

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-

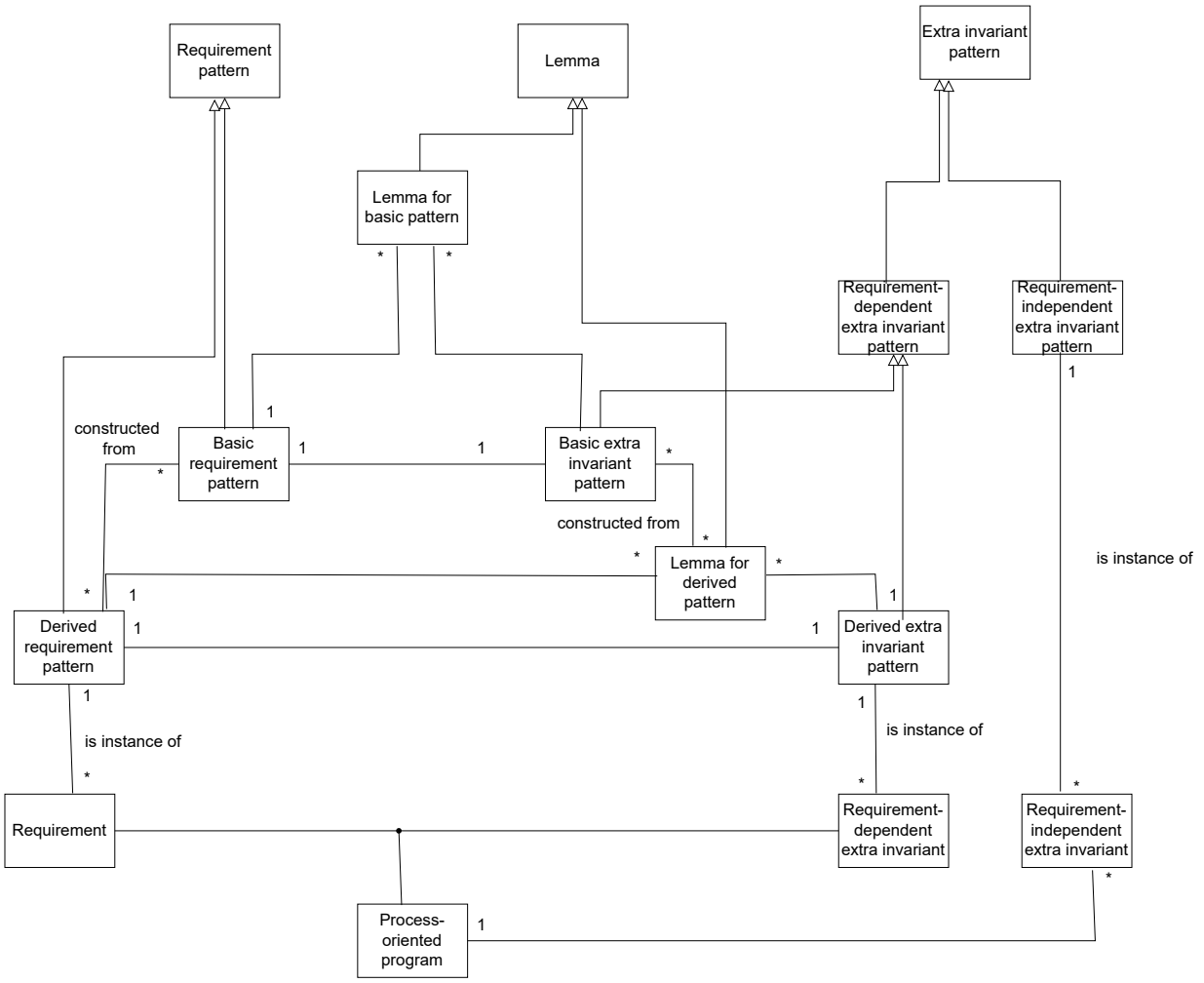

Fig. 1. Relationship between different kinds of patterns and lemmas.

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

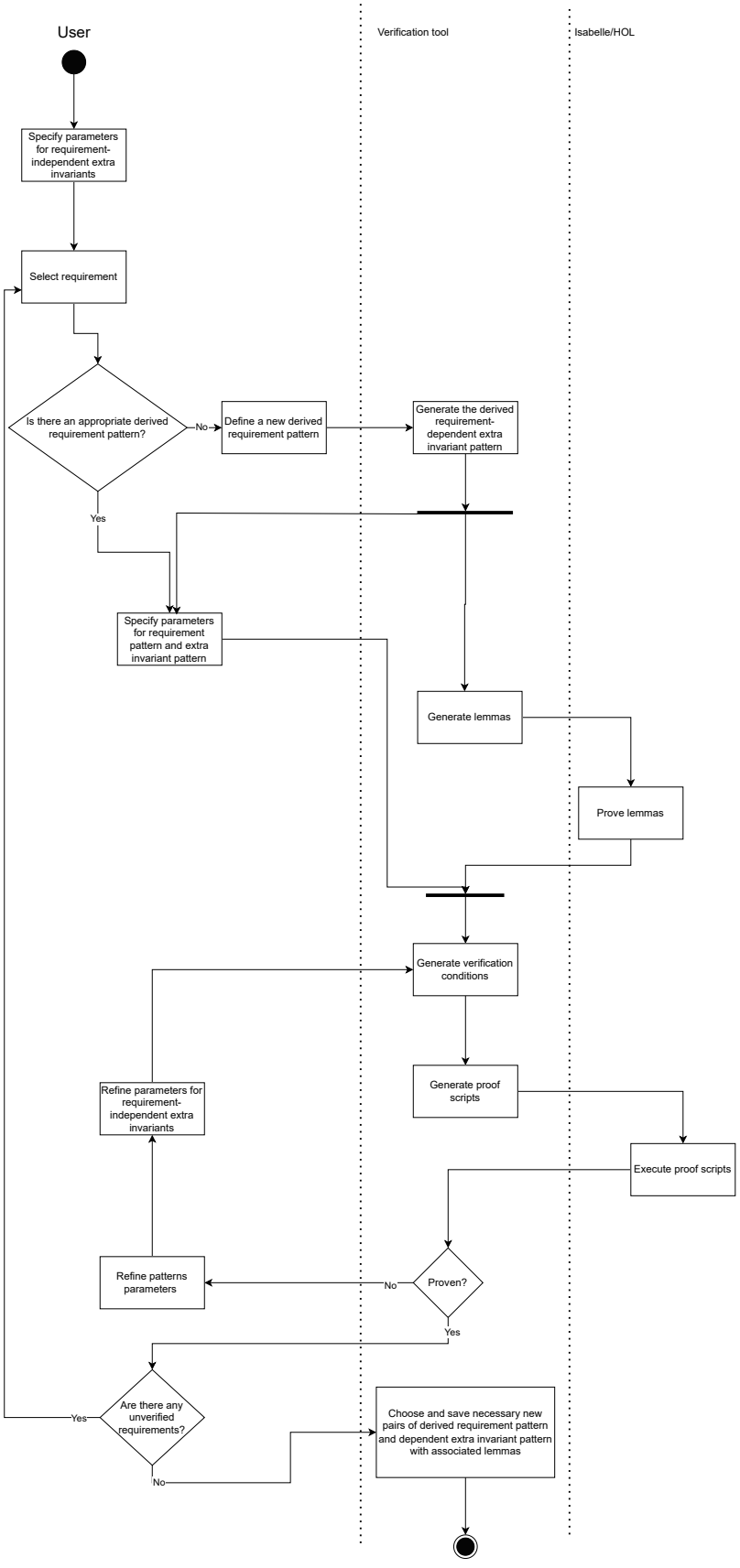

Fig. 2. User interaction with the verification tool.

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.

```
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.

The hand dryer control program is presented on Figure 3:

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, $s_1 \dots s_n$, r, $r_1 \dots r_m$ 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.

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

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.

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

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:

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

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

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:

$A_1 \equiv \lambda(s, r_1).r_1[hands] = False;$
$A_2 \equiv \lambda(s, r_2).r_2[dryer] = True \wedge r_2[hands] = False;$
$A_3 \equiv \lambda(s, r_3).r_3[dryer] = False \vee r_3[hands] = True;$
$t \equiv 10;$
$t_1 \equiv \lambda s.\ \mathbf{if}\ \ getPstate(s, Ctrl) = drying\ \ \mathbf{then}\ \ ltime(s, Ctrl)\ \ \mathbf{else}\ \ 10.$

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 itera-

tively 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

UDK 004.4'42

An Exact Schedulability Test for Real-time Systems with an Abstract Scheduler

Garanina N.O. (A.P. Ershov Institute of Informatics Systems SB RAS)

In this paper, we formally describe real-time systems with an abstract scheduler using Kripke structures. This formalization allows us to refine the abstract scheduler in its terms. We illustrate this approach with a non-preemptive global fixed priority scheduler (NE-GFP). We also formulate a safety property for real time systems using linear temporal logic LTL. We implement our formalization of real-time systems with a NE-GFP scheduler in language Promela used in the SPIN verification tool and make experiments for proving or disproving the safety property to evaluate the effectiveness of our approach.

Keywords: real-time systems, exact schedulability test, Kripke structures, model checking, Promela, SPIN

1. Introduction

Classical real-time systems proposed in [11] are a set of tasks that occur from time to time with period P or more, have a deadline D and execution time C. In the modern world, such systems arise literally at every step – these are embedded systems, and Internet of Things systems, and technological processes, business processes, and automatic control systems in the automotive industry, avionics, space industry, etc. These tasks can use the resources of one or more processors. As a rule, there are significantly fewer processors than tasks, so the question of scheduling their execution naturally arises. There are different ways to specify schedulers depending on the subject domain. For example, in some cases it is possible to allow interruption of low-priority tasks, while in other cases such an approach can lead to a system failure. The main question for real time systems described in terms of execution time, deadline and periodicity is the question of safety: is it true that in a given set of tasks with fixed features and a given scheduler, no task will ever miss its deadline?

This problem has been solved for many years for various real time systems and various schedulers. For systems with only one processor, the problem has been studied quite well [11]. However, for multiprocessor systems, the problem of a very large number of task behavior variants arises, and exact methods for checking the safety property turn out to be poorly applicable in an explicit form. Approaches based on near-optimal scheduling have been proposed [2], but

exact schedulers are preferable and exact approaches continue to develop [3, 4, 8, 12, 17].

In addition to the development of specialized methods for precise schedulability verification, there is a small number of works using general formal methods for analyzing programs and systems. In particular, [9, 16] proposes the representation of static-priority global multiprocessor scheduling and non-Preemptive self-suspending real-time tasks using timed automata, which make the verification of the safety of real time systems rather resource-consuming. In [13], a special case of a real time system was modeled in the Promela language of the SPIN verification tool [7], and in [15] authors present a Promela model for real-time system on a single-processor. In [6], graph games is used to make easier the reachability problem in exact shedulability test.

In our work, we formally represent real-time systems with an abstract scheduler as Kripke structures, which are used to verify parallel and distributed systems using model checking. Such formalization allows us to refine the abstract scheduler in terms of the proposed Kripke structure and obtain specific real-time systems. We present an example of refining the abstract scheduler to a non-preemptive GFP scheduler. In addition, we formulate the safety property in terms of liner temporal logic LTL [5]. We model our formalization of real-time systems with a non-preemptive GFP scheduler in the Promela language of the SPIN and conduct a series of experiments to prove or disprove the safety property to clarify the performance of our method.

The rest of the paper has the following structure. In Section 2, we recall base definitions of real-time systems and scheduling and formalise real-time systems as Kripke structures. Section 3 considers features of the Promela and describes the Promela model for a real-time system with the non-preemptive global fixed priority scheduler. The conclusion is given in Section 4.

2. A Real-time Kripke Structure

We consider that *a real-time system* is a set of tasks $T = (T_1, ..., T_n)$, where each *task* $T_i = (C_i, D_i, P_i)$ has *an execution time* C_i, *a relative deadline* D_i, and *a minimum period* P_i. Each task $T_i \in T$ can generate a potentially infinite number of jobs, every of which requires C_i units of time. These jobs must be completed before D_i time units after the release time. Release time instants are separated by at least P_i time units. If there are no other restrictions on jobs' releases, these tasks are refereed as *sporadic tasks*. In this paper, we study the base case of real-time systems in which all task parameters are integers. All jobs are executed on m processors. If the number processors is less then number of tasks ($m < n$), we need a *scheduler* to decide which task's job to run next. We assume that scheduling decisions are taken at

discrete time instants starting from 0. The *schedulability test problem* is to detect if every jobs of every task in the real-time system is finished before its deadline. For the rest of the paper we fix above real-time system T.

We would like to deal with real-time system T as with a finite model to apply model checking techniques. Inspired by the paper [1], we introduce current values of task parameters as follows. For every task $i \in T$, let tuple $s_i = (i, C'_i, D'_i, P'_i, rel_i, bad_i)$ be a state of task i, where

- $C'_i \leq C_i$ is time left until a job of this task ends;
- $D'_i \leq D_i$ is time until the deadline of a job of this task;
- $P'_i \leq P_i$ is time until the the next admissible release of a job of this task;
- $rel_i \in \mathbb{B}$ is boolean variable which marks a job release: it becomes *true* when task i release a job, and it becomes $false$ when the job is completed.
- $bad_i \in \mathbb{B}$ is boolean variable which marks that a job misses the deadline soon: it is $false$ if $C'_i \leq D'_i$, and it becomes *true* otherwise.

For brevity, we refer to boolean constants *true* and $false$ as **1** and **0**, respectively. For representing the processors' load, we also introduce variable *busy*: a number of jobs currently executing at some moment ($busy \leq m$).

A scheduler must exactly define conditions under which each task can execute its job and conditions for permitting a job execution. There are many types of schedulers, for example,

- Global fixed priority (GPF). The set of tasks are ordered: T_1 has the highest priority, T_n has the lowest priority and a major task job has priority over a minor task job;
- Earliest deadline first (EDF). The task with the closest deadline has the highest priority;
- Non-preemptive. No job can be interrupted by other job (even from a major task);
- Preemptive. A job can be interrupted by other job from a major task.

In our Kripke structure, for modeling an abstract scheduler we use an abstract predicate $go(i, s, t)$ that is *true* if task i can execute the job at state t which is a successor of state s, and $false$ otherwise. Further, we specify $go(i, s, t)$ for the non-preemptive GPF scheduler.

For calculating the change of processor load *busy*, we also need to compute a modification number – a quantity of tasks which jobs are just finished or released and just admitted to execution by the scheduler. For this, we introduce predicate $fin(i, s)$ for just finished jobs that is *true* if task i finishes its job in state s, i.e. $s.C'_i = 0$, and $false$ otherwise. To compute the modification number, we treat $go(i, s, t)$ and $fin(i, s)$ as integer numbers (1 for *true* and 0 for $false$).

Let $Prop$ be a set of propositions consisting of arithmetic comparisons of current values of task parameters and propositions about a number of running jobs.

Now, we define real-time system T with an abstract scheduler as *a real-time structure* $M^T = (S^T, s_0^T, R^T, L^T)$, where

- the finite set of states $S^T = \prod_{i=1}^{n}(\{i\} \times [0..C_i] \times [0..D_i] \times [0..P_i] \times \mathbb{B} \times \mathbb{B}) \times [0..m]$; for global state $s \in S^T$, $s_i = (i, C'_i, D'_i, P'_i, rel_i, bad_i)$ is *a projection of* s *on task* i, and $s.C'_i$, $s.D'_i$, $s.P'_i$, $s.rel_i$, $s.bad_i$ and $s.busy$ are projections of s on its components;
- the initial state $s_0^T = \prod_{i=1}^{n}\{(i, C_i, D_i, P_i, \mathbf{0}, \mathbf{0})\} \times \{0\}$;
- the total transition relation $R^T \in S^T \times S^T$ is defined by composing relations for i-task projections of global states s and t. $(s,t) \in R^T$ iff $t.busy = s.busy + \sum_{i=1}^{n}(go(i,s,t) - fin(i,s))$ and one of the following points holds:
  1. $s_i = (i, C_i, D_i, P_i, \mathbf{0}, \mathbf{0})$, and
     (a) $t_i = (i, C_i, D_i, P_i, \mathbf{0}, \mathbf{0})$ – task i do nothing;
     (b) $t_i = (i, C_i, D_i - 1, P_i - 1, \mathbf{1}, \mathbf{0})$, and $\neg go(i,s,t)$ – task i releases the job and it is not started;
     (c) $t_i = (i, C_i - 1, D_i - 1, P_i - 1, \mathbf{1}, \mathbf{0})$, and $go(i,s,t)$ – task i releases the job and it is immediately started;
  2. $s_i = (i, C'_i, D'_i, P'_i, \mathbf{1}, \mathbf{0})$, $t_i = (i, C'_i, D'_i - 1, P'_i - 1, \mathbf{1}, \mathbf{0})$ with $s.D'_i > 0$, and $\neg go(i,s,t)$ – task i is waiting for permitting its job;
  3. $s_i = (i, C'_i, D'_i, P'_i, \mathbf{1}, \mathbf{0})$, $t_i = (i, C'_i - 1, D'_i - 1, P'_i - 1, \mathbf{1}, \mathbf{0})$ with $0 < s.C'_i < C_i$, $0 < s.D'_i < D_i$, $s.C'_i \leq s.D'_i$, and $go(i,s,t)$ – task i executes a job;
  4. $s_i = (i, 0, D'_i, P'_i, \mathbf{1}, \mathbf{0})$, or $s_i = (i, C_i, D'_i, P'_i, \mathbf{0}, \mathbf{0})$, or $s_i = (i, C_i, D_i, P'_i, \mathbf{0}, \mathbf{0})$, and
     (a) if $s.P'_i = 0$
        i. $t_i = (i, C_i - 1, D_i - 1, P_i - 1, \mathbf{1}, \mathbf{0})$, and $go(i,s,t)$ – task i finishes its job normally, and it is releasing and starting its job at this moment;
        ii. $t_i = (i, C_i, D_i - 1, P_i - 1, \mathbf{1}, \mathbf{0})$, and $\neg go(i,s,t)$ – task i finishes its job normally, and it is releasing and not starting its job at this moment;
        iii. $t_i = (i, C_i, D_i, P_i, \mathbf{0}, \mathbf{0})$ – task i finishes its job normally, and it is not releasing;
     (b) if $s.D'_i = 0$ and $s.P'_i > 0$ then $t_i = (i, C_i, D_i, P'_i - 1, \mathbf{0}, \mathbf{0})$ – task i finishes its job normally and waiting for the next release;
     (c) if $s.D'_i > 0$ then $t_i = (i, C_i, D'_i - 1, P'_i - 1, \mathbf{0}, \mathbf{0})$ – task i finishes its job normally and waiting for the next release;

5. $s_i = (i, C'_i, C'_i - 1, P'_i, \mathbf{1}, \mathbf{0})$ and $t_i = (i, C'_i, C'_i - 1, P'_i, \mathbf{0}, \mathbf{1})$ – a job of task i will miss the deadline definitely;
6. $s_i = (i, C'_i, C'_i - 1, P'_i, \mathbf{0}, \mathbf{1})$ and $t_i = (i, C'_i, C'_i - 1, P'_i, \mathbf{0}, \mathbf{1})$ – task i is in the bad state forever.

- evaluation function $L : Prop \longrightarrow 2^{S^T}$ is standard: it assigns comparison propositions to those states in which they are true.

To refine the abstract scheduler, we need to specify predicate $go(i, s, t)$. This predicate for all types of schedulers uses information about the load of processors *busy* and the system tasks: their parameters, priorities, time from releases, time until deadlines, etc. All this information is available at the system states s and t, hence $go(i, s, t)$ can be formulated in terms of our real-time Kripke structures. For example, in the case of non-preemptive global fixed priority scheduler, predicate $go(i, s, t) = (|Maj_i| + s.busy < m)$, where $Maj_i = \{j \in [1..n] \mid j < i \wedge ((s.rel_j = \mathbf{1} \wedge s.C_j = C_j) \vee (s.rel_j = \mathbf{0} \wedge t.rel_j = \mathbf{1}))\}$ is a set of released jobs with higher priority that have not yet started.

We mark a state of real-time system T with $D'_i = C'_i - 1$ for some task i as *a bad state* because in this case there is no time left to meet the deadline. Bad states in the Kripke structure M^T is a set $Bad\_States = \{s \in S^T \mid \exists i \in [1..n] : \ s.bad_i = \mathbf{1}\}$. Let proposition *bad* be *true* in state s if $\bigvee_{i=0}^{n}(s.bad_i = \mathbf{1})$ is *true*, and be $false$ otherwise. Hence, the exact schedulability test for real-time system T is to check if LTL formula $\Phi_T = \mathbf{G}(\neg bad)$ is satisfiable in M^T: the real-time systems never reaches a state in which some task in its bad state.

3. A Real-time System with a Non-preemptive Global Fixed Priority Scheduler in Promela

In this section, we describe implementation of real-time structure M^T for real-time system T in Promela – an input language of model checker SPIN. Promela language is used to describe parallel communicating processes based on the CSP formalism [10]. Promela program consists of parallel processes communicating through channels or shared variables. The execution of a set of Promela parallel processes exploits the interleaving semantics. Interleaving can be bounded by `atomic` and `d_step` statements, which permit interruption of specified sequence of process actions. The Promela language includes blocking control-flow statements `if` and `do`. Promela model can be verified by model checker SPIN against LTL requirements, hence it assumes only finite types for model variables. Our real-time structure M^T has a finite number

of states and its representation in Promela does not require data abstractions.

We model M^T in Promela to perform exact schedulability test, i.e. to detect if every jobs of every task is completed before its deadline. We specify an abstract scheduler of M^T as a non-preemptive global fixed priority scheduler. For simplicity, we also set the period P_i be equal to D_i for every i. Below, we give the implementation details skipping some code for brevity.

The initial Promela process starts the scheduler and `NumPrc` number of tasks with synthetic parameters C_i and D_i naming each task by its number i. Specific non-synthetic real-time systems can be modelled by operator `run task(i, C_i, D_i)` for particular C_i and D_i.

```
1  init{
2  atomic{
3    run scheduler();
4    for (i : 0 .. NumPrc-1){ run task(i, i+1, 2*(i+2)); }}
5  }
```

Listing 1: Promela-process for launching the system

Following the definition of M^T, we describe tasks that release and execute their jobs as Promela processes that implement transition relation R^T almost directly. For example, the Promela code for Point 2 of the definition is given in Listing 2 (lines 8–20). In green comments of this listing, we give explanations of the task actions. If a task fails its deadline, it sets special boolean variable `BAD` to *true* (line 26).

```
1  proctype task (byte me; byte C; byte D) {
2   bool go = false;       // predicate go(i,s,t) computed by the scheduler
3   bool release = false; // a variable for marking job realeases
4   byte C_cur = C;        // C' -- time left until a job ends
5   byte D_cur = D;        // D' -- time until the deadline
6  do
7  :: atomic{ C_cur == C && D_cur == D && !release -> // can release a job
8     if
9     :: release = true;    // releases a job
10       request ! me       // requests the scheduler for job execution
11       work++;            // action for synchronization step
12       responce[me] ? go // receives the responce from the scheduler
13       if
14       ::  go ->
15           C_cur--; D_cur--; busy++; // executes the job, point 2.c
16       :: else -> D_cur--;           // waits, point 2.b
17       fi
18    :: work++; // do nothing, point 2.a
19    fi
20    responce[me] ? _ // action for synchronization step
21    }
22 :: atomic{ C_cur == C && D_cur > 0 && D_cur < D && C_cur <= D_cur && release
       -> // release and not started
23   ...
24 :: atomic{ C_cur > D_cur && release -> // fails deadline
25    BAD = true; // point 6
26    break;
27    }
28 od
29 }
```

Listing 2: Promela-process for tasks

In our model, the scheduler gives permission for job executions and synchronises Promela processes for tasks to prevent unwanted interleaving. First, the scheduler collects requests for job executions in boolean array `req_prc` (lines 4–6). Second, following non-preemptive and GFP settings, the scheduler knowing the number of free processors (line 10) scans the array of requests from its first element in the order (line 11). If it finds that the next in line task i asks for its job execution and there are free processors (line 13), it sends signal *go* to this task (lines 14) and decreases the number of free processors (lines 15). When the number of free processors becomes zero (line 16), the scheduler sends the rest of requesting tasks the message with the prohibition of its job execution (line 17). After scheduling actions, this Promela process performs synchronization between tasks, resetting the number of `work` tasks executing their step and sending them permission to continue working (line 24). The scheduler's scanning, communication and synchronisation actions are placed in an `atomic` block, the sequence of actions of which is treated by SPIN as a single computational step. This placing provides synchronisation between the tasks and the scheduler and decreases the model checking computational complexity.

```
1  proctype scheduler(){
2  ...
3  do
4  :: atomic{ work > 0 && !end -> ...
5     request ? num; req_prc[num] = true; // collects requests
6     ... }
7  :: atomic{ work == NumPrc && empty(request) ->  ...
8     if
9     :: NumReqs > 0 ->      // there are some requests
10       free = MAX - busy; // number of free processors
11       for (i : 0 .. NumPrc - 1){
12       if
13       :: req_prc[i] && free != 0 ->
14          responce[i] ! true; ...  // go!
15          free--;
16       :: req_prc[i] && free == 0 ->
17          responce[i] ! false; ... // don't go!
18        :: else -> skip;
19        fi
20        } ...
21    :: else -> skip; // no requests
22    fi
23    // actions for synchronization step:
24    work = 0; for (i : 0 .. NumPrc - 1){ responce[i] ! true }
25    }
26 :: BAD -> break;
27 od
28 }
```

Listing 3: Promela-process for the scheduler

To exactly test a real-time system for schedulability, we check LTL formula `[]!BAD` in SPIN tool. If this formula is satisfiable in the real-time system, no its task misses its deadline. We made some experiments with SPIN model checker (version 6.5.1), on a CPU with 4 cores

and 8 GB RAM. We definitely prove the safety of our real-time system for 5 tasks and 4 processors. When we increase the number of tasks by one, the SPIN verification time becomes too long (more then an hour) for proving safety. But if a real-time system is unsafe, SPIN discovers it very quickly: for 40 tasks and 20 processors it takes only 0.074 seconds to find the counterexample (the sequence of task releases leading to missing some deadline).

The complete Promela model for real-time systems with the non-preemptive global fixed priority scheduler is located in the repository [18].

4. Conclusion

In this paper, we formalise real-time systems with an abstract scheduler as a Kripke structure – the real-time Kripke structure. We show that this abstract scheduler can be refined in terms of a real-time Kripke structure. In particular, we present a non-preemptive global fixed priority scheduler by specifying conditions under which tasks' jobs are started.

In future, we plan to refine the abstract scheduler for other types of schedulers, i.e. preemptive GPF scheduler, preemptive and non-preemptive EDF scheduler, etc. These specifications can used for modeling real-time systems in input languages of model checkers in order to perform exact schedulability tests and for teaching. We also plan to use our formalisation of real time systems to develop new effective algorithms for the exact schedulability test, e.g. backtracking based algorithms.

References

1. V. Bonifaci and A. Marchetti-Spaccamela, Feasibility analysis of sporadic real-time multiprocessor task systems. // Algorithmica, 2012.
2. B. B. Brandenburg and M. Gul, Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. // Proceedings of Real-Time Systems Symposium (RTSS). IEEE, 2016.
3. Artem Burmyakov, Enrico Bini, and Eduardo Tovar. 2015. An exact schedulability test for global FP using state space pruning. // In Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS '15). Association for Computing Machinery, New York, NY, USA, 225–234. https://doi.org/10.1145/2834848.2834877
4. A. Burmyakov, E. Bini and C. -G. Lee, Towards a Tractable Exact Test for Global Multiprocessor Fixed Priority Scheduling. // in IEEE Transactions on Computers, vol. 71, no.

11, pp. 2955-2967, 1 Nov. 2022, doi: 10.1109/TC.2022.3142540.

5. Clarke E.M., Henzinger T. A., Veith H., and Bloem R. Handbook of model checking. Springer, 2018. 1210 p.
6. G. Geeraerts, J. Goossens and T. -V. -A. Nguyen, A Backward Algorithm for the Multiprocessor Online Feasibility of Sporadic Tasks. // 2017 17th International Conference on Application of Concurrency to System Design (ACSD), Zaragoza, Spain, 2017, pp. 116-125, doi: 10.1109/ACSD.2017.9.
7. Holzmann G. J. The SPIN Model Checker, Primer and Reference Manual. Addison-Wesley: 2003.
8. Pourya Gohari, Jeroen Voeten, and Mitra Nasri. Reachability-Based Response-Time Analysis of Preemptive Tasks Under Global Scheduling. In 36th Euromicro Conference on Real-Time Systems (ECRTS 2024). Leibniz International Proceedings in Informatics (LIPIcs), Volume 298, pp. 3:1-3:24, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2024) https://doi.org/10.4230/LIPIcs.ECRTS.2024.3
9. Guan, N., Gu, Z., Deng, Q., Gao, S., Yu, G. Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking. // Software Technologies for Embedded and Ubiquitous Systems. SEUS 2007. Lecture Notes in Computer Science, vol 4761. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-75664-4_26
10. Hoare C. A. R. Communicating sequential processes. Prentice-Hall: 1985.
11. Jane W. S. Liu, Real-Time Systems. Prentice Hall, 2001. 610 p.
12. Ranjha, S., Gohari, P., Nelissen, G. et al. Partial-order reduction in reachability-based response-time analyses of limited-preemptive DAG tasks. Real-Time Syst 59, 201–255 (2023). https://doi.org/10.1007/s11241-023-09398-x
13. Staroletov, S. A Formal Model of a Partitioned Real-Time Operating System in Promela. // Proceedings of the Institute for System Programming of the RAS (2020): Volume 32, Issue 6, Pages 49–66, DOI: https://doi.org/10.15514//ISPRAS-2020-32(6)-4
14. Sun, Y., Lipari, G. A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor Global Fixed-Priority scheduling. // Real-Time Syst 52, 323–355 (2016). https://doi.org/10.1007/s11241-015-9245-9
15. P. Sukvanicha, A. Thongtak and W. Vatanawood, Formalizing Real-Time Embedded System into Promela // MATEC Web of Conferences 35 03003 (2015) DOI: 10.1051/matecconf/20153503003

16. B. Yalcinkaya, M. Nasri and B. B. Brandenburg, An Exact Schedulability Test for Non-Preemptive Self-Suspending Real-Time Tasks. // 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 2019, pp. 1228-1233, doi: 10.23919/DATE.2019.8715111.
17. Q. Zhou, G. Li, C. Zhou and J. Li, Limited Busy Periods in Response Time Analysis for Tasks Under Global EDF Scheduling. // in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 40, no. 2, pp. 232-245, Feb. 2021, doi: 10.1109/TCAD.2020.2994265
18. URL: https://github.com/GaraninaN/RealTimeSystems/blob/main/np-gfp-rts.pml, 2024.

UDK 004

Policy Based Interval Iteration for Probabilistic Model Checking

Mohagheghi M. (Vali-e-Asr University of Rafsanjan, Rafsanjan, Iran)

Khademi A. (Vali-e-Asr University of Rafsanjan, Rafsanjan, Iran)

Reachability probabilities and expected rewards are two important classes of properties that are used in probabilistic model checking. Iterative numerical methods are applied to compute the underlying properties. To guarantee soundness of the computed results, the interval iteration method is used. This method utilizes two vectors as the upper-bound and lower-bound of values and uses the standard value iteration method to update the values of these vectors. In this paper, we use a combination of value iteration and policy iteration to update these values. We use policy iteration to update the values of the lower bound vector. For the upper-bound vector, we use a modified version of value iteration that marks useless actions to disregard them for the remainder of the computations. Our proposed approach brings an opportunity to apply some advanced techniques to reduce the running time of the computations for the interval iteration method. We consider a set of standard case studies and the experimental results show that in most cases, our proposed technique reduces the running time of computations.

Keywords: Probabilistic model checking, Sound iterative methods, Policy iteration

1. Introduction

Value iteration (VI) and policy iteration (PI) are two well-known iterative numerical methods that are used to approximate the required values in the analysis of Markov decision processes. In reinforcement learning, VI and PI are used to compute the optimal expected rewards (or costs) for an intelligent agent. In formal verification, these two methods are also used to compute the optimal reachability probabilities: the minimal or maximal probability of reaching a goal state. In both cases, linear programming (LP) can be used to compute exact values. Because of the scalability of this technique, LP can not be used for large models. VI and PI are used instead to approximate the required values. These techniques compute iteratively until satisfying a stopping criterion[1, 5]. In reinforcement learning, a discount factor is considered to guarantee the convergence of computations. The iterative computations terminate when the maximal difference between two consecutive iterations drop below a threshold [5, 6]. In formal

verification, where the optimal probabilities or expected rewards until reaching a goal state is needed, some pre-computations are applied to disregard those states for which, the accumulated expected rewards converge to infinite. For the remaining states, VI or PI can be applied. Both approaches iteratively update state-values until reaching the case where the difference between values drops the threshold. Although in most cases, this stopping criterion is sufficient to verify the correctness of a given specified property, there is not any guarantee for the precision of the computed values. There are some cases in probabilistic model checking where VI or PI terminate and the reported reachability probability (or expected reward) is far away from the exact value [2, 9]. To overcome this deficiency, the interval iteration method has been proposed as an extension to the standard value iteration and is implemented in most model checker tools. In interval iteration (II), two vectors of values are considered as an upper-bound and lower-bound of the state-values. These vectors are updated iteratively until reaching a case where the maximal difference of state values between these two vectors drop the threshold. In such case, it is guaranteed that the exact value is between the two ones and maximal error is controlled [4, 9, 15].

The running time of the standard iterative methods for probabilistic model checking is an important challenge that affects their application on large models. While the number of states of a model can grow exponentially in the number of components, that causes the state explosion problem, a modern model checker should control the total number of numerical computations. Several approaches have been proposed to accelerate the VI and PI methods. The results of [13] demonstrate that the modified and improved versions of PI can outperform VI for most standard case study models. For the interval iteration method, using two vectors of values brings more computations and the running time is more than the VI and PI methods. For this method, it is important to utilize improved techniques to reduce the overall running time. In most previous works, VI is mainly used to update the values of these vectors, i.e., in each iteration, VI is used to update the upper-bound and lower-bound values of each state. Appropriate state ordering for VI has been used as an approach to accelerate the interval iteration method [8]. This approach is applied in [2, 15] to reduce the overall running time of the computations. To the best of our knowledge, no other work has considered more advanced techniques to accelerate the iterative computations for the II method. In this paper, we focus on the problem of accelerating II and select PI as the standard method for updating the state-values of this method. We utilize several accelerating methods for PI to improve the performance of II. The main benefit of these

accelerating methods is that in most iterations, they disregard less important computations and focus on more important ones.

1.1. Related work

Value iteration and policy iterations have been used as the standard iterative methods in probabilistic model checking [1, 3, 5]. However, the soundness of these methods was studied for the first time in [7] where a counterexample is proposed that shows VI may terminate and return a result that is far away from the exact value. To cover this problem and guarantee the soundness of the computed values, the interval iteration method is proposed in [7] for the extremal reachability probabilities. The extension of II for the extremal expected rewards is proposed in [2] where some approaches are suggested for computing the upper-bound vector of values. In some cases, the computed upper-bounds are far away from true values that causes a large number of iterations in the computations of the II method. To reduce the overall running times and start from better upper-bounds, the sound value iteration[15] and optimistic value iteration[10] methods are proposed for computing the minimal and maximal expected rewards. The possibility of applying policy iteration for the interval iteration method is studied in [9]. It has only considered the standard PI method and do not studied the impact of modified policy iteration or other improved technique on the performance of computations.

2. Preliminaries

In this section, we review some important concepts of probabilistic model checking. More details are available in [2, 3, 8]. For a finite set S and vectors $\underline{x} = (x_s)_{s\in S} \in \mathbb{R}^{|S|}$ and $\underline{y} = (y_s)_{s\in S} \in \mathbb{R}^{|S|}$, we write $\underline{x} \leq \underline{y}$ if $x_s \leq y_s$ for all $s \in S$ and we write $\underline{x} < \underline{y}$ if $x_s < y_s$ for all $s \in S$.

2.1. Markov Decision Process

Definition 1. A Markov Decision Process (MDP) is a tuple $M = (S, s_0, Act, P, R)$ where:

- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- Act is a finite set of actions.
- $P : S \times Act \times S \to [0, 1]$ is a probabilistic transition function such that for each state s and the action $\alpha \in Act$ we have $\sum_{s'\in S} P(s, \alpha, s') \in \{0, 1\}$.

- $R : S \times Act \to \mathbb{R}$ is a reward function.

An action α is *enabled* in state s, if $\sum_{s' \in S} P(s, \alpha, s') = 1$. The operational semantics of MDP M is as follows. M is initiated with state s_0. Assume that M is in state s. First, one of the enabled actions of s is selected non-deterministically. According to the selected action α, the reward $R(s, \alpha)$ is collected by the system. Then due to probabilistic transition function P, one of the α-successor states of s (a state for which there is a transition from s via action α) is chosen. That is, s' is the next state with probability $P(s, \alpha, s')$. A discrete-time Markov chain (DTMC) is an MDP in which every next state is selected just probabilistically, i.e. it has exactly one enabled action [1].

The minimal (or maximal) probability of reaching one of the goal states is called extremal reachability probability. Extremal expected rewards are defined as the minimal (or maximal) expectation of accumulated rewards until reaching a goal state. For a state $s \in S$, we use $\mathbb{E}_s^{max}$ to denote the maximal expected reward. Some graph-based methods are exploited to detect the set of states that the extremal reachability probability is one [1, 3]. These sets are used to computing the maximal or minimal expected rewards [8].

A standard way to resolve non-deterministic choices in MDPs is to define and use policies. A deterministic policy (also called adversary) is function $\pi : S \to Act$ that maps each state of the MDP to one of its enabled actions. A policy is memory less where it decide the actions only based on the last visited state [8]. For unbounded reachability probabilities and expected rewards, the optimal action of each state $s \in S$ is determined based of the probability distribution of the outgoing transitions and the computed values of the successor states and it is not important which states have been visited before reaching s. Hence, for these classes of properties, that are the topic of the current work, it is sufficient to consider deterministic and memory-less policies [1, 3].

Iterative numerical methods are used to approximate the values of the expected rewards. Value iteration, as a well known method uses a vector x^k to store the approximated values of the maximal expected rewards in iteration k. The values of x_s^k determines the approximated value of $\mathbb{E}_s^{max}$ after k iterations. In each iteration k the value of x^k is computed according to the computed value of x^{k-1} from the previous iteration. Theoretically, value iteration can finally converge to the exact expected values, that is, $\lim_{k\to\infty} x_s^k = \mathbb{E}_s^{max}$; but practically, a convergence criterion is exploited to terminate the iterations. To do so, the maximum difference of computed values between two consecutive iterations are compared with a threshold ϵ in which the value

iteration method terminates when $max_{s \in S^1_{min}}(x^k_s - x^{k-1}_s) < \epsilon$ [8]. In the literature, several methods are proposed to accelerate the standard iterative methods using state prioritizing approaches [8, 13]. Policy iteration (PI) is another option for approximating the extremal reachability probabilities or expected reward[5]. This method follows a set of rounds. In each round, PI considers a policy to select one action for each state. Based on the selected actions a DTMC is induced and a set of iterations is applied to update the state values. For the first round, the policy is defined by randomly selecting the actions of each state. For the other rounds, policies are defined by considering the best action of each state by considering the computed values of the previous round. The computations continue until reaching a stable policy, where the last two policies are the same [9].

2.2. Interval Iteration for Expected Accumulated Rewards

Value iteration with the standard termination criterion can not guarantee the precision of approximated values [6]. Interval iteration methods are proposed to guarantee the precision of computed values for the extremal reachability probabilities [4, 6] and for extremal expected rewards [2]. Two vectors $\underline{x}$ and $\underline{y}$ are utilized to approximate the lower and upper bound of the extremal expected reward values. In this case, vectors x and y can finally converge from below and above to $\underline{\mathbb{E}}^{max}$.

Considering ϵ for the precision of computations, the interval iteration method iterates until the maximum difference of values between two vectors for all states drops below 2ϵ, that is in an iteration k, if $\max_{s \in S^1_{min}}(y^k_s - x^k_s) < 2\epsilon$ holds. After termination we have $|\frac{y^k_s + x^k_s}{2} - \mathbb{E}^{max}_s| < \epsilon$ for each state $s \in S^1_{min}$. Algorithm 1 computes the interval iteration for the maximal expected rewards. To avoid some redundant computations, a modified version (called separate interval iteration) is proposed in [12].

Several methods are available for pre-computation to calculate the starting values for $\underline{x}$ and $\underline{y}$. In the case of non-negative rewards, $\underline{0}$ is trivially a starting point of $\underline{x}$. For $\underline{y}$, several techniques are presented in [2] to calculate the starting point. A main drawback of these techniques to compute an initial value for the vector $\underline{y}$ is that in come cases, the proposed initial values are far away from the exact values. This drawback increases the running time of computations. To have better initial values for $\underline{y}$ several techniques have been proposed in [10, 12, 15]. The idea of these techniques is to consider the computed values for the vector $\underline{x}$ to approximate the initial value for $\underline{y}$. In the proposed approaches in [10, 12], the standard value iteration

Algorithm 1 Interval iteration for $\mathbb{E}_s^{max}$.

input: an MDP $M = (S, s_0, Act, P, R)$, a set G of goal states, the set S^1_{min}, a threshold ϵ and two initial vectors $\underline{x}$ and $\underline{y}$ for the lower and upper bound of values

output: Approximation of $\mathbb{E}_s^{max}$ for all $s \in S$ with the precision of ϵ

repeat

for all $s \in S^1_{min}$ **do**

$x_s = \max\limits_{\alpha \in Act(s)} (R(s,\alpha) + \sum_{s' \in S} P(s,\alpha,s') \cdot x_{s'})$;

$y_s = \min\{y_s, \max\limits_{\alpha \in Act(s)} (R(s,\alpha) + \sum_{s' \in S} P(s,\alpha,s') \cdot y_{s'})\}$;

end for

until $max_{s \in S^1_{min}}(y_s - x_s) \leq 2\epsilon$;

return $(\frac{y_s + x_s}{2})_{s \in S^1_{min}}$;

method is applied to update the values of $\underline{x}$. After terminating value iteration, the initial value for $\underline{y}$ is computed by considering the computed values for $\underline{x}$. In this step and for each state $s \in S$, the proposed approach considers $y_s = c \cdot x_s$ where $c > 1$ is a constant. To check the soundness of the computed vector $\underline{y}$ several number of iterations of the VI method is applied on this vector. If after these iterations, the value of all states decrease, it is guaranteed that the computed vector is correctly an upper-bound of the values and can be used for the remainder computations. Otherwise, several other iterations are applied on the vector $\underline{x}$ and a higher value for c is considered. The process continues until getting a sound vector $\underline{y}$. More details about this approach is available in [10].

3. The Proposed Method

The possibility of using PI to update the values of the vectors $\underline{x}$ and $\underline{y}$ for the interval iteration method has been recently investigated in [9]. It is not trivial to apply the PI method to update the values of the upper-bound vector and [9] proposes an example where the PI method picks some non-optimal policies and the II method with it terminates with some values far away from the exact one. This case may happen for both extremal reachability probabilities and expected rewards. Instead, a hybrid approach is mentioned in [9] to use a combination of VI, PI and action elimination. The experimental evaluation of [9], however, does not cover this case and only reports the results for the optimistic version of II[10] where the standard VI is used to update values. It also considers PI when it uses II to resolve each induced DTMC. In

this section, we provide our approach to apply PI as a solver for the interval iteration method and discuss the possibility of applying some advanced techniques to reduce the overall running time.

3.1. Sound Interval Iteration with Policy Iteration and Action Elimination

In this section, we explain how to use PI and VI with action elimination for the interval iteration method in a sound way. We propose this approach in Algorithm 2. It gets a MDP model M, a constant $K > 1$, a threshold ϵ for terminating the computations and an initial value for the vector $\underline{x}$. The threshold ϵ can be considered as is in VI, PI and II[5]. For the initial value of $\underline{x}$ the values for all states can be set to 0. For better explanation, we divide the algorithm to three steps. The first step, contains a while loop and follows an approach similar to the optimistic VI to compute a sound vector for upper-bound. However, our algorithm applies PI to update the states of the vector $\underline{x}$. After terminating PI, the initial value for the vector $\underline{x}$ is computed by considering the constants $c > 1$ and $d > 0$. Then the algorithm updates the values of $\underline{y}$ by applying K iterations of VI. If the value of all states decrease, the current values for $\underline{y}$ provide a correct upper bound. Otherwise, several additional iterations of PI is applied by considering smaller values for the threshold ϵ. It is guaranteed that these process will finally terminate while has computed a sound vector $\underline{y}$ for upper-bounds [10, 12].

The second step, for each state $s \in S$ considers the computed values for x_s and label those actions that give under-approximation for y_s. Such actions can not give a correct value for the upper-bound and should be disregarded for the remainder of the computations.

The third step updates the value of the vectors $\underline{x}$ and $\underline{y}$ until reaching a point where the difference of the upper-bound and lower-bound for all states is less than ϵ. It uses PI to update the values of the lower-bound and VI with non-disabled actions for the upper-bound.

Using PI to update the values of $\underline{x}$ and mark some actions as disabled can bring several opportunity to improve the performance of the II method. There are some cases where the standard version of PI outperform VI and hence, the former method can be a better option than the later. In addition, the modified and improved versions of PI as are discussed in [9, 12, 13] are faster than VI in most cases. Eliminating non-optimal actions avoids useless computations and reduces the overall running time of the computations. The soundness of this algorithm relies on the fact that PI will always converge to the true values from below if it uses sufficient

Algorithm 2 II with Policy Iteration for $\mathbb{E}_s^{max}$.

input: an MDP $M = (S, s_0, Act, P, R)$, a set G of goal states, the set S^1_{min}, a threshold ϵ and an initial vectors $\underline{x}$ for the lower bound of values

output: Approximation of $\mathbb{E}_s^{max}$ for all $s \in S$ with the precision of ϵ

flag = True;

while flag == True **do**

Apply PI on the vector $\underline{x}$ considering ϵ for termination of computations.

for all $s \in S$ **do**

$y_s = c \cdot x_s + d$;

end for

iters = 1;

while iters < K **do**

for all $s \in S$ **do**

$y_s = \min\{y_s, \max_{\alpha \in Act(s)} (R(s,\alpha) + \sum_{s' \in S} P(s,\alpha,s') \cdot y_{s'})\}$;

end for

iters = iters + 1;

end while

if $\exists\ s \in S^1_{min}$ where y_s has never decreased **then**

$\epsilon = \epsilon/2$;

else

flag = false;

end if

end while

for all $s \in S^1_{min}, \alpha \in Act(s)$ **do**

if $x_s > R(s,\alpha) + \sum_{s' \in S} P(s,\alpha,s') \cdot y_{s'}$ **then**

Mark α as disabled.

end if

end for

repeat

for all $s \in S^1_{min}$ **do**

Update x_s using PI;

Update y_s considering non-disabled Actions;

end for

until $max_{s \in S^1_{min}}(y_s - x_s) \leq 2\epsilon$;

return $(\frac{y_s + x_s}{2})_{s \in S^1_{min}}$;

number of iterations, i.e., it never gets stuck in wrong state values. For the upper-bound the soundness is provided when we apply the standard VI method [15] or disregard those actions that do not surely give the optimal values.

4. Experimental Results

To analyse the running time of the proposed method in this paper with the standard one, we consider a set of standard case studies that are widely used in the previous works [5, 9]. We use the PRISM model checker [11] to run the experiments. We also implemented Algorithm 2 in PRISM. To do so, we consider the interval iteration [2] method (called II), sound value iteration [15] (called SVI) and optimistic value iteration wthod[10] (called OVI) from the previous works. We also consider Algorithm 2 with the standard PI method (PII), its modified version [14] (MPII) and its improved version (IPII)[13]. The results of the experiments are reported in Table1. For each method, we report the running time of iterative computations and the total number of computations. All reported times are in seconds.

Table 1

The running time of the evalutaed methods.

Model Name	Parameter	Number of	II		SVI		OVI		PII		MPII		IPII	
(parameters)	Values	states	time	iters	time	iters	time	iters	time	iterations				
Coin	4,6	63616	89.7	67791	48.6	37750	41.3	30123	42	42056	33	30158	32.8	30147
(K,N)	4,10	104576	475	191473	226	93589	184	78495	154	85345	138	78538	139	78545
	6,2	1258240	832	20811	509	13840	475	11624	663	31445	339	11642	342	11675
Zeroconf	6	798471	23.2	1491	12.6	747	10.7	635	11.7	647	11.5	1086	9.78	794
(K)	10	3001911	96	1546	49.6	850	40.9	695	38.8	695	39,7	695	40.2	697
	14	4427159	174	1822	76.6	812	69.7	746	63.1	746	67.2	782	749	64
Wlan	200	171542	15.4	6502	12.2	4709	11	4391	11.3	6092	9.24	4394	9.75	4719
(ttm)	800	409142	102	16509	96.2	15117	93.7	14668	85.4	17464	77.3	14671	78.6	14742
	1600	725942	426	35841	367	31482	343	29016	293	32003	286	29019	288	29022

In most cases, our proposed method with modified policy iteration works better than the optimistic value iteration method. In some cases, even the proposed method with the standard policy iteration is faster than OVI. Altough in these cases, the overall number of iterations increases, but because they consider only one action in most iterations, they are faster than OVI. There are also some cases where the proposed method with improved policy iteration outperforms the other cases.

5. Conclusion

In this work, we propose an approach to use PI for updating the values of states for the II method. Experimental results demonstrate that our proposed technique with modified policy

iteration outperforms the best known previous work for optimistic value iteration. For the future works, we plan to consider several improvements for policy iteration to reduce the overall running time of the proposed technique.

References

1. Baier, Christel, and Joost-Pieter Katoen. Principles of model checking. MIT press, 2008.
2. Baier, Christel, Joachim Klein, Linda Leuschner, David Parker, and Sascha Wunderlich. "Ensuring the reliability of your model checker: Interval iteration for Markov decision processes." In International Conference on Computer Aided Verification, pp. 160-180. Cham: Springer International Publishing, 2017.
3. Baier, C., Hermanns, H., Katoen, JP. (2019). The 10,000 Facets of MDP Model Checking. In: Steffen, B., Woeginger, G. (eds) Computing and Software Science. Lecture Notes in Computer Science, vol 10000. Springer, Cham. DOI: 10.1007/978-3-319-91908-9_21
4. Br?zdil, Tom??, Krishnendu Chatterjee, Martin Chmelik, Vojt?ch Forejt, Jan K?et?nsk?, Marta Kwiatkowska, David Parker, and Mateusz Ujma. "Verification of Markov decision processes using learning algorithms." In Automated Technology for Verification and Analysis: 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings 12, pp. 98-114. Springer International Publishing, 2014.
5. Vojt?ch Forejt, Marta Kwiatkowska, Gethin Norman and David Parker. Automated Verification Techniques for Probabilistic Systems. In M. Bernardo and V. Issarny (editors), Formal Methods for Eternal Networked Software Systems (SFM'11), volume 6659 of LNCS, pages 53-113, Springer. June 2011.
6. Haddad, Serge, and Benjamin Monmege. "Interval iteration algorithm for MDPs and IMDPs." Theoretical Computer Science 735 (2018): 111-131. DOI: 10.1016/j.tcs.2016.12.003
7. S. Haddad and B. Monmege. Reachability in MDPs: Refining convergence of value iteration. In 8th International Workshop on Reachability Problems (RP), volume 8762 of Lecture Notes in Computer Science, pages 125137. Springer, 2014.
8. Kwiatkowska, Marta, David Parker, and Hongyang Qu. "Incremental quantitative verification for Markov decision processes." In 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN), pp. 359-370. IEEE, 2011.
9. Hartmanns, Arnd, Sebastian Junges, Tim Quatmann, and Maximilian Weininger. "A practitioners guide to MDP model checking algorithms." In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 469-488. Cham: Springer Nature Switzerland, 2023.
10. Hartmanns, Arnd, and Benjamin Lucien Kaminski. "Optimistic value iteration." In International Conference on Computer Aided Verification, pp. 488-511. Cham: Springer International Publishing, 2020.
11. Kwiatkowska, M., Norman, G., and Parker, D. (2011a). Prism 4.0: Verification of probabilistic real-

time systems. In International conference on computer aided verification, pages 585591. Springer
12. Mohagheghi, Mohammadsadegh, and Khayyam Salehi. "Accelerating Interval Iteration for Expected Rewards in Markov Decision Processes." In ICSOFT, pp. 39-50. 2020.
13. Mohagheghi, Mohammadsadegh, Jaber Karimpour, and Ayaz Isazadeh. "Improving modified policy iteration for probabilistic model checking." Computer Science 23, no. 1) (2022): 63-80.
14. Puterman, Martin L., and Moon Chirl Shin. "Modified policy iteration algorithms for discounted Markov decision problems." Management Science 24, no. 11 (1978): 1127-1137.
15. Quatmann, Tim, and Joost-Pieter Katoen. "Sound value iteration." In International Conference on Computer Aided Verification, pp. 643-661. Cham: Springer International Publishing, 2018.