Propositional Projection Temporal Logic Specification Mining

Nan Zhang, Xiaoshuai Yuan, and Zhenhua Duan(✉)

Institute of Computing Theory and Technology, and ISN Laboratory, Xidian University, Xi'an 710071, China
nanzhang@xidian.edu.cn, yuanxiaoshuai@stu.xidian.edu.cn, zhhduan@mail.xidian.edu.cn

Abstract. This paper proposes a dynamic approach of specification mining for Propositional Projection Temporal Logic (PPTL). To this end, a pattern library is built to collect some common temporal relation among events. Further, several algorithms of specification mining for PPTL are designed. With our approach, PPTL specifications are mined from a trace set of a target program by using patterns in the library. In addition, a specification mining tool PPTLMiner supporting this approach is developed. In practice, given a trace set and user selected patterns, PPTLMiner can capture PPTL specifications of target programs.

Keywords: Propositional projection temporal logic · Pattern · Trace · Specification mining

1 Introduction

A software system specification is a formal description of the system requirements. Formal languages are often employed to write specifications so as to prevent the ambiguity written in natural languages. The common used formal languages include Temporal Logic (TL) and Finite State Automata (FSA). Software system specification can be used to test and verify the correctness and reliability of software systems [13]. However, due to various kinds of reasons, a great number of software systems lack formal specifications. In particular, for most of legacy software systems, formal specifications are missed. This makes the maintenance of software systems difficult. To fight this problem, various kinds of specification mining approaches are proposed [10–12, 14, 15, 17, 19–21].

Walkinshaw et al. [19] present a semi-automated approach to inferring FSAs from dynamic execution traces that builds on the QSM algorithm [8]. This algorithm infers a finite state automaton by successively merging states. Lo et al.

This research is supported by National Key Research and Development Program of China under Grant No. 2018AAA0103202, National Natural Science Foundation of China under Grant Nos. 61751207 and 61732013, and Shaanxi Key Science and Technology Innovation Team Project under Grant No. 2019TD-001.

W. Wu and Z. Zhang (Eds.): COCOA 2020, LNCS 12577, pp. 289–303, 2020.
https://doi.org/10.1007/978-3-030-64843-5_20

propose Deep Specification Mining (DSM) approach that performs deep learning for mining FSA-based specifications [11]. FSA specifications are intuitive and easily to be used for verifying and testing programs. However, most of FSA specification mining approaches suffer from accuracy and correctness for representing properties of programs. Yang et al. [21] present an interesting work on mining two-event temporal logic rules (i.e., of the form $G(a \rightarrow XF(b))$, where G, X and F are LTL operators, which are statistically significant with respect to a user-defined "satisfaction rate". Wasylkowski et al. [20] mine temporal rules as Computational Tree Logic (CTL) properties by leveraging a model checking algorithm and using concept analysis. Lemieux et al. [12] propose an approach to mine LTL properties of arbitrary length and complexity. Similar to the above research work, most of specification mining approaches employ LTL and CTL as the property description languages. Due to the limitation of the expressiveness of LTL and CTL, some temporal properties such as periodic repetition properties cannot be characterized.

Since the expressiveness of Propositional Projection Temporal Logic (PPTL) is full regular [3,18], in this paper, we propose a dynamic approach to mining PPTL properties based on a pattern library. PPTL contains three primitive temporal operators: next ($\bigcirc$), projection (prj) and chop-plus ($+$). Apart from some common temporal properties that can be formalized in LTL and CTL, PPTL is able to describe two other kinds of properties: interval sensitive properties and periodic repetition properties. With the proposed approach, we abstract API/method calls as events. A trace is a sequence of API/method calls occurred during program execution. Daikon [1] is used to generate raw traces first, then a tool *DtraceFilter* we developed is employed to further refine the traces. Patterns are used to characterize common temporal relations among events. Two categories of patterns, Occurrence and Order, are used. These patterns are predefined in a pattern library. The proposed mining algorithms require two inputs: an instantiated pattern formula P and a refined execution trace τ. To obtain an instantiated pattern formula, we need to specify a pattern formula which can be either a user-defined one or a predefined one in the library. The pattern is instantiated by substituting atomic propositions with concrete events. After pattern instantiation, several mining algorithms based on PPTL normal form [3–7] are employed to recursively check whether τ satisfies P.

The contribution of the paper is three-fold. First, we propose a PPTL temporal rule specification mining approach so that full regular properties can be mined. Second, we develop a tool PPTLMiner which supports the proposed mining approach. Third, we build a pattern library to cover all common patterns accumulated from literatures and abstracted from the existing software systems. The library is open, user-editable and in constant expansion and growth.

This paper is organized as follows. In the next section, PPTL is briefly introduced. In Sect. 3, the trace generation and the construction of the pattern library are presented. In Sect. 4, the overall framework of PPTLMiner and key algorithms are elaborated. Finally, conclusions are drawn in Sect. 5.

2 Propositional Projection Temporal Logic

In this section, we briefly introduce our underlying logic, Propositional Projection Temporal Logic (PPTL), including its syntax and semantics. It is used to describe specifications of programs. For more detail, please refer to [3,7].

Syntax of PPTL. Let *Prop* be a set of atomic propositions and $p \in Prop$. The syntax of PPTL is inductively defined as follows.

$$P ::= \ p \mid \bigcirc P \mid \neg P \mid P \vee Q \mid (P_1, ..., P_m)\ prj\ Q \mid P^+$$

where $P_1, ..., P_m, P$ and Q are well-formed PPTL formulas. Here, $\bigcirc$ (next), prj (projection) and $+$ (chop-plus) are primitive temporal operators.

Semantics of PPTL. Let $B = \{true, false\}$ and N be the set of non-negative integers. Let ω denote infinity. PPTL formulas are interpreted over intervals. An interval σ is a finite or infinite sequence of states, denoted by $\sigma = \langle s_0, s_1, \ldots \rangle$. A state s_i is a mapping from *Prop* to B. An interpretation $\mathcal{I} = (\sigma, k, j)$ is a subinterval $\langle s_k, \ldots, s_j \rangle$ of σ with the current state being s_k. An auxiliary operator $\downarrow$ is defined as $\sigma \downarrow (r_1, \ldots, r_m) = \langle s_{t_1}, s_{t_2}, \ldots, s_{t_n} \rangle$, where $t_1, \ldots, t_n$ are obtained from $r_1, \ldots, r_m$ by deleting all duplicates. That is, $t_1, \ldots, t_n$ is the longest strictly increasing subsequence of $r_1, \ldots, r_m$. The semantics of PPTL formulas is inductively defined as a satisfaction relation below.
(1) $\mathcal{I} \models p$ **iff** $s_k[p] = true$.
(2) $\mathcal{I} \models \bigcirc P$ **iff** $(\sigma, k+1, j) \models P$.
(3) $\mathcal{I} \models \neg P$ **iff** $\mathcal{I} \not\models P$.
(4) $\mathcal{I} \models P \vee Q$ **iff** $\mathcal{I} \models P$ or $\mathcal{I} \models Q$.
(5) $\mathcal{I} \models (P_1, \ldots, P_m)\ prj\ Q$ **iff** there exist m integers $k = r_0 \leq r_1 \leq \ldots \leq r_m \leq j$ such that $(\sigma, r_{l-1}, r_l) \models P_l$ for all $1 \leq l \leq m$, and one of the following two cases holds:
- if $r_m = j$, there exists r_h such that $0 \leq h \leq m$ and $\sigma \downarrow (r_0, \ldots, r_h) \models Q$;
- if $r_m < j$, then $\sigma \downarrow (r_0, \ldots, r_m) \cdot \sigma_{(r_m+1..j)} \models Q$.

(6) $\mathcal{I} \models P^+$ **iff** there exist m integers $k = r_0 \leq r_1 \leq \ldots \leq r_m = j$ $(m \in N)$ such that $(\sigma, r_{l-1}, r_l) \models P$ for all $1 \leq l \leq m$.
Derived Formulas. Some derived formulas in PPTL are defined in Table 1.

Operator Priority. To avoid an excessive number of parentheses, the precedence rules shown in Table 2 are used, where 1 = highest and 9 = lowest.

Definition 1 (PPTL Normal Formal). Let Q be a PPTL formula and Q_p denote the set of atomic propositions appearing in Q. Q is in normal form if Q has been rewritten as

$$Q \equiv \bigvee_{j=1}^{n_0} (Q_{ej} \wedge \epsilon) \vee \bigvee_{i=1}^{n} (Q_{ci} \wedge \bigcirc Q'_i)$$

where $Q_{ej} \equiv \bigwedge_{k=1}^{m_0} \dot{q_{jk}}$, $Q_{ci} \equiv \bigwedge_{h=1}^{m} \dot{q_{ih}}$, $l = |Q_p|$, $1 \leq n \leq 3^l$, $1 \leq n_0 \leq 3^l$, $1 \leq m \leq l$, $1 \leq m_0 \leq l$; $q_j k, q_i h \in Q_p$, for any $r \in Q_p$, $\dot{r}$ means r or $\neg r$; Q'_i is a general PPTL formula.

Table 1. Derived formulas

A1	$\varepsilon \stackrel{\text{def}}{=} \neg \bigcirc true$	A2	$more \stackrel{\text{def}}{=} \bigcirc true$
A3	$\bigcirc^0 P \stackrel{\text{def}}{=} P$	A4	$\bigcirc^n P \stackrel{\text{def}}{=} \bigcirc(\bigcirc^{n-1} P)(n > 0)$
A5	$\odot P \stackrel{\text{def}}{=} \varepsilon \vee \bigcirc P$	A6	$P; Q \stackrel{\text{def}}{=} (P, Q)\ prj\ \varepsilon$
A7	$\Diamond P \stackrel{\text{def}}{=} true; P$	A8	$\Box P \stackrel{\text{def}}{=} \neg\Diamond\neg P$
A9	$len(n) \stackrel{\text{def}}{=} \bigcirc^n \varepsilon$	A10	$skip \stackrel{\text{def}}{=} len(1)$
A11	$P^* \stackrel{\text{def}}{=} P^+ \vee \varepsilon$	A12	$P \parallel Q \stackrel{\text{def}}{=} (P; true) \wedge Q \vee P \wedge (Q; true)$
A13	$fin \stackrel{\text{def}}{=} \Diamond\varepsilon$	A14	$inf \stackrel{\text{def}}{=} \neg fin$

Table 2. Operator priority

1.	$\neg$	2.	$+, *$	3.	$\bigcirc, \odot, \Diamond, \Box$
4.	$\wedge$	5.	;	6.	$\vee$
7.	prj	8.	$\parallel$	9.	$\rightarrow, \leftrightarrow$

In some circumstances, for convenience, we write $Q_e \wedge \epsilon$ instead of $\bigvee_{j=1}^{n_0}(Q_{ej} \wedge \epsilon)$ and $\bigvee_{i=1}^{r}(Q_i \wedge \bigcirc Q_i')$ instead of $\bigvee_{i=1}^{n}(Q_{ci} \wedge \bigcirc Q_i')$. Thus,

$$Q \equiv (Q_e \wedge \epsilon) \vee \bigvee_{i=1}^{r}(Q_i \wedge \bigcirc Q_i')$$

where Q_e and Q_i are state formulas. The algorithm of translating a PPTL formula into its normal form can be found in [4–6].

3 Pattern Library Construction and Trace Generation

Our specification mining algorithm relies on two inputs: a pattern and a program execution trace. A pattern is a property template in which the atomic proposition symbols need to be instantiated as events (namely, API or method calls) occurred during program execution. A trace is a sequence of method calls in the execution of a program. In this section, we present how to build the pattern library and traces.

3.1 Pattern and Pattern Library

Patterns are abstracted from common software behaviors and used to describe occurrence of events or states during program execution [9]. A pattern is a logical representation of certain event relation. The *APIs* and methods in a target program are defined as events. We say that an event occurs whenever it is called in the execution of the program. In the following, we define a quadruples to represent and store patterns.

Definition 2 (Pattern). A pattern $T =< C, N, R, A >$ is a tuple where C is a pattern category indicating occurrence or order of events, N a pattern name, R a PPTL formula, and A an annotation.

Following Dwyer et al.'s SPS [9] and Autili et al.'s PSP framework [2], we also classify patterns into two categories, *Occurrence* and *Order*.

The *Occurrence* category contains 18 patterns that indicate presence or absence of certain events or states during program execution. For instance, (1) *Absence* means that an event never happens; (2) *Universality* indicates that an event always occurs during program execution; (3) *Existence* shows that an event occurs at least once during program execution; and (4) *Bounded Existence* tells us that an event has a limited number of occurrences during program execution, e.g. event $f.open()$ occurs twice.

The *Order* category contains 19 patterns that represent relative temporal orders among multiple events or states occurred during program execution. For example, (1)"*s precedes p*" indicates that if event p occurs, event s definitely occurs before p; (2) "*s responds p*" means that if event p occurs, event s definitely occurs after p; (3)*Chain* (s, t) means that a combination chain of events s and t. (s, t) *precedes* p means that if event p happens, chain events (s, t) certainly happen before p, and (s, t) *responds* p means that if event p happens, (s, t) certainly responds to p [2,9].

Pattern Library. A pattern library L is a set containing all patterns p we collected. After an in-depth investigation of the existing literature and programs specified behavior characteristics, we build a pattern library and some patterns are shown in Table 3 and Table 4.

Table 3. Pattern library - occurrence category

No.	Pattern Name	PPTL Formula	Annotation
1	Universality	$\Box p$	Event p always occurs
2	Absence	$\Box \neg p$	Event p never occur
3	Existence	$\Diamond p$	Event p occurs at least once
4	Frequency	$\Box \Diamond p$	Event p occurs frequently
5	Both Occur	$\Diamond p \wedge \Diamond q$	Events p and q both occur
6	Simultaneity	$\Diamond (p \wedge q)$	Events p and q occur at the same time
7	Prefix of Trace	$\Box \Diamond p; more$	Event p occurs frequently at a prefix of a trace
8	Suffix of Trace	$\Diamond \Box p$	Event p occurs continuously at a suffix of a trace

3.2 Trace Generation

We concern only specifications of temporal relations among the methods or API calls occurred during program execution.

Table 4. Pattern library - order category

No.	Pattern Name	PPTL Formula	Annotation
1	Precedence (1-1)	$\Diamond p \rightarrow (\Box\neg p; s)$	Event s takes precedence over event p
2	Response (1-1)	$\Box(s \rightarrow \bigcirc\Diamond p)$	Event p responds to event s
3	Until	$(\Box p; \bigcirc s) \vee s$	Event p occurs until event s occurs
4	Response Invariance	$\Box(p \rightarrow \bigcirc\Box s)$	If p has occurred, then in response s holds continually
5	Chop	$\Box p; \bigcirc\Box q$	There exists a time point t such that event p occurs continuously before t and event q continuously after t
6	Never Follow	$\Box(p \rightarrow \odot\,\Box\neg q)$	Event p is never followed by event q

Definition 3 (Trace). A trace is a sequence of methods or API calls (namely events) with parameters.

Example 1. A trace of a program using stack structure.
trace $\tau_1 = \langle StackAr(int), isFull(), isEmpty(), top(), isEmpty(), topAndPop(), isEmpty(), isFull(), isEmpty(), top(), isEmpty(), push(java.lang.Object), isFull()\rangle$

Example 2. A trace of a program manipulating files.
trace $\tau_2 = \langle open(f1), write(f1), read(f1), close(f1), open(f2), delete(f1), read(f2), write(f2), write(f2), read(f2), close(f2), delete(f2)\rangle$

We use Daikon [1] as an auxiliary tool to generate traces. Daikon can dynamically detect program invariants. A program invariant is a property that remains unchanged at one or more positions of program execution. The common invariants are APIs, functions, global or local variables, arguments, return values and so on. Invariants can be used to analyze behavior of a program. Dynamic invariant detection refers to a process of running a program so as to check variables and assertions detected in the program execution [16].

Daikon generates a sequence containing all invariants and stores it in a *dtrace* file in which the invariants are stored line by line. The program execution traces we need are contained in this sequence. Since there exists an amount of redundant information, the *dtrace* file needs to be further refined.

The whole process of generating a trace is shown in Fig. 1.

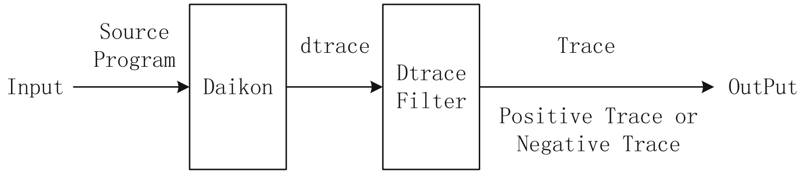

Fig. 1. The process of trace generation

Step 1. Generating sequences of program invariants

A source program and its arguments are input to Daikon so that a sequence of program invariants is generated. The sequence is written in a file *f.dtrace* in the *dtrace* format. When the program is executed with different arguments for a desired number n of times, we obtain a set $Pool_1 = \{f_i.dtrace | i = 1, \ldots, n\}$ of program traces.

Step 2. Filtering of sequences of program invariants

A filter tool *DtraceFilter* has been developed to filter out redundant information, including parameters, variables, return values and useless spaces, in each file $f_i.dtrace$ of $Pool_1$. As a result, sequences consisting of only *APIs* and method calls constitute a new set $Pool_2 = \{f_i.trace | i = 1, \ldots, n\}$.

Step 3. Parsing traces in $Pool_2$

Each trace $f_i.trace$ in $Pool_2$ needs to be parsed so as to obtain a API/method-name list $f_i.event$. These lists constitute a set $Event = \{f_i.event | i = 1, \ldots, n\}$.

Step 4. Optimizing traces in $Pool_2$

We can specify desired API/method names from the lists in *Event* according to the requirements. *DtraceFilter* can be used to select the events we concern from each list in *Event* to build a positive list $f_i.pevent$ of events, and generate a set $PositiveEvent = \{f_i.pevent | i = 1, \ldots, n\}$.

Based on *PositiveEvent*, *DtraceFilter* further refines each $f_i.trace$ in $Pool_2$ to get a positive trace $f_i.ptrace$ consisting of only the events in $f_i.pevent$, and obtain a set $PositiveTrace = \{f_i.ptrace | i = 1, \ldots, n\}$.

We can also specify undesired API/method names from the lists in *Event*. In a similar way, *DtraceFilter* can be used to build a negative list $f_i.nevent$ of events and generate $NegativeEvent = \{f_i.nevent | i = 1, \ldots, n\}$. After deleting the negative events from each trace $f_i.trace$ in $Pool_2$, *DtraceFilter* builds a set $NegativeTrace = \{f_i.ntrace | i = 1, \ldots, n\}$.

4 PPTL Specification Mining

Based on the Pattern Library and set of refined traces presented in the previous section, an approach to PPTL specification mining is proposed and a specification mining tool, PPTLMiner, is developed. In this section, the framework of PPTLMiner and some key algorithms are presented in detail.

4.1 The Framework of PPTLMiner

The integrated design of PPTLMiner is shown in Fig. 2. It consists of the following six parts.

(1) Pattern Library. The Pattern Library covers all patterns we obtain after investigating literatures and programs. Our Pattern Library is open, user-editable and in constant expansion and growth. New patterns can be inserted into the library from time to time. For more details, refer to Sect. 3.1.

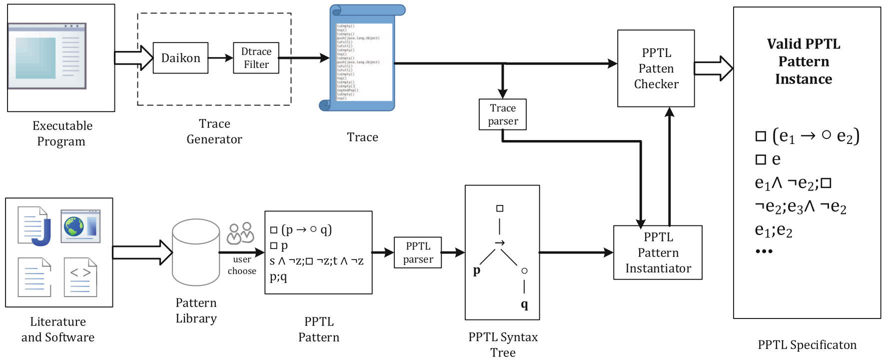

Fig. 2. The framework of PPTLMiner

(2) Trace Generator. The function of the Trace Generator is to generate traces from an executable program. To do so, an executable program and its arguments are input into Daikon to produce raw traces (dtrace files). Then *Dtracefilter* is employed to filter out redundant information in dtrace files to obtain trace files, which are further refined to obtain positive and negative traces. For more details, refer to Sect. 3.2.

(3) PPTL Parser. The input of PPTL Parser is a PPTL formula. PPTL Parser is developed by means of Flex and Bison. It can be used to generate a PPTL syntax tree for any PPTL formula.

(4) Trace Parser. The function of Trace Parser is two-fold. The first is to parse traces generated by the Trace Generator and restore them in an appropriate data structure so that the traces can conveniently be used by PPTL Pattern Checker. The second is to calculate a set $E = \{e_1, e_2, \ldots, e_n\}$ of events appeared in the traces so as to instantiate PPTL patterns.

(5) PPTL Pattern Formula Instantiator. The instantiator requires two inputs: (a) a PPTL pattern formula P, and (b) E, the set of events produced by Trace Parser. The function of the instantiator is to instantiate a pattern formula P by substituting atomic propositions in P by events in $Events$.

(6) PPTL Pattern Checker. PPTL Pattern Checker also requires two inputs: (a) a trace τ produced by Trace Generator, and (b) an instantiated pattern formula P generated by PPTL Pattern Formula Instantiator. The function of the Checker is to decide whether trace τ satisfies P.

4.2 Mining Process and Algorithms

In this subsection, we present the mining process and algorithms in detail.

(1) Syntax Tree of PPTL Formula

By syntax analysis, a PPTL Pattern Formula is parsed into a syntax tree. A syntax tree consists of a root node and two child nodes. The root node is of two attributes, NODETYPE and STRING, which indicate the type and name of the root node, respectively. All nodes having two null child nodes in the syntax tree of a PPTL pattern formula P constitute a set $S(P)$ of atomic propositions. For instance, for an atomic proposition p, its NODETYPE is "atomic proposition" while its STRING is "p". Two child nodes are all null. For formula $P_1; P_2$, its NODETYPE is "chop" while its STRING is ";". It has two non-null child nodes, $child_1$ and $child_2$, where $child_1$ is the root of P_1 while $child_2$ is the root of P_2. $S(P_1; P_2) = S(P_1) \cup S(P_2)$. More Examples are shown in Fig. 3.

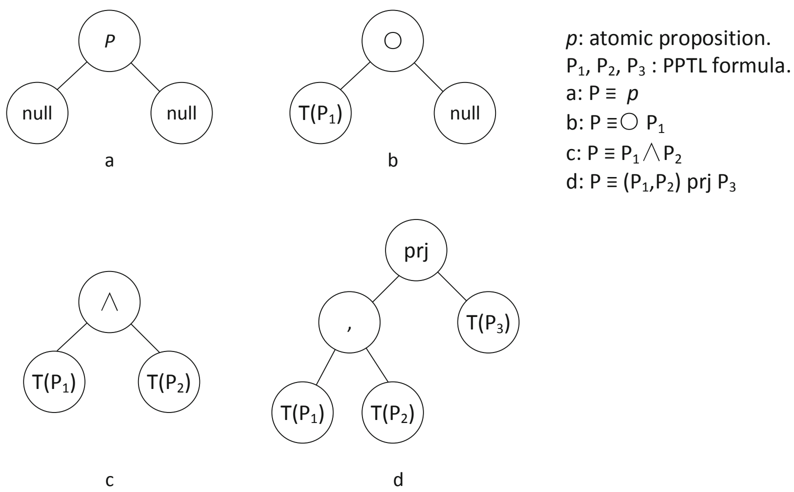

Fig. 3. PPTL syntax tree

(2) Instantiating PPTL Pattern Formulas

Based on the set $S(P)$ of atomic propositions and set E collected by Trace Parser, a PPTL Pattern Formula P is instantiated by Algorithm 1.

(3) PPTL Pattern Check

We use Algorithm 2, Algorithm 3 and Algorithm 4 to check whether τ satisfies Q, where τ is a refined trace generated in Sect. 3.2 while Q is an instantiated PPTL pattern formula obtained in part (2). These algorithms are based on PPTL Normal Form.

In particular, Algorithm 2, i.e. CheckBasedonNF(P, τ), first checks the satisfiability of P. If P is satisfiable, it is translated into its normal form P_{nf} by calling the existing external function $NF(\cdot)$ given in [3]. Then Algorithm 3 NFCheckTrace(P_{nf}, τ) is called to decide whether τ satisfies P_{nf}.

In function NFCheckTrace(P_{nf}, τ), the first disjunct $P_{nf}.child_1$ is first checked. If NFCheckTrace($P_{nf}.child_1, \tau$) is true, P_{nf} is already satisfied by τ. Otherwise the rest disjuncts $P_{nf}.child_2$ are further checked.

Algorithm 1. function Instantiator(E, S, P)

Input: E: a set of events;
Input: S: a set of atomic propositions appearing in P;
Input: P: a syntax tree of a PPTL pattern formula;
Output: P_s: a set of instantiated PPTL pattern formulas.

```
begin
P_s ← null;
m is a patttern instance;
/* m = {(ap_i, ep_i) | ap_i ∈ S & ep_i ∈ E & 1 ≤ i ≤ |S| & ap_i ≠ ap_j if i ≠ j)} */
M is a set of pattern instances; /* M = {m_1, m_2, ....} */
M ← null;
Count is used for count the number of m;
Count ← 0;
/* (E.size())!/(E.size()-S.size())! is the total number of non-duplicate pattern instances */
while Count <= (E.size())!/(E.size()-S.size())! do
    E_1 is a set used to store ep ∈ E has been checked;
    m ← null;
    E_1 ← null;
    for all ap in S do
        while true do
            ep is an event randomly selected from E;
            if ep not in E_1 then
                m.insert(ap, ep); /* ap is mapped to ep */
                E_1.insert(ep); /* ep is labeled */
                break;
            end if
        end while
    end for/* build m */
    if m not in M then
        M.insert(m);
        count++;
    end if
end while/* build M */
for all m in M do
    P_ins is a copy of P; /* P_ins is used for instantiation */
    P_ins ← P;
    for all node in P_ins do
        if node.type == AtomicProp then
            for all m_i in m do
                if m_i.ap == node.name then
                    node.name ← m_i.ep;
                end if
            end for
        end if
    end for
    P_s.insert(P_ins); /* insert pattern instance P_ins into set P_s */
end for
return P_s
end
```

Algorithm 2. function CheckBasedonNF(P, τ)

Input: P: An instantiated PPTL pattern formula;
Input: τ: A program execution trace;
Output: $True$ if τ satisfies P, $False$ otherwise.

```
begin
q is a boolean variable;
q = CheckSatisfiability(P); /* check satisfiability of P [5] */
if ¬q then
    return False;
else
    P_nf = NF(P); /* transform P into its normal form [5] */
    return NFCheckTrace(P_nf,τ);
end if
end
```

To check a disjunct, two cases need to be considered: (1)$P_e \wedge \varepsilon$ and (2)$P_c \wedge \bigcirc P_f$. For the first case, the function checks whether τ satisfies P_e and whether τ is empty. If both are true, $P_e \wedge \varepsilon$ is satisfied by τ. For the second case, the function checks whether τ satisfies P_c and whether $tail(\tau)$ satisfies P_f. If both are true, $P_c \wedge \bigcirc P_f$ is satisfied by τ. In checking whether τ satisfies the state formula P_e or P_c, Algorithm 4 StateFormulaCheck(P_s,τ) is called. For doing so, function StateFormulaCheck(P_s,τ) is simply to check the satisfiability of state formula P_s over τ by considering several syntax constructs of P_s.

Algorithm 3. function NFCheckTrace(P_{nf}, τ)

Input: P_{nf}: A PPTL formula in its normal form;
Input: τ: A program execution trace;
Output: *True* if τ satisfies P_{nf}, *False* otherwise.

```
begin
τ_bak = τ;
switch P_nf.type do
    case OrProp
        q_1 is a boolean variable;
        q_1 = NFCheckTrace(P_nf.child_1, τ);
        if q_1 then /* first disjunct is satisfied by τ */
            return True;
        else/* select another disjunct */
            τ = τ_bak;
            return NFCheckTrace(P_nf.child_2, τ);
        end if
    case AndProp
        P_c = P_nf.child_1; /* if P_nf.child_2 is ε, P_c stands for P_e */
        q_2 is a boolean variable;
        q_2 = StateFormulaCheck(P_c, τ); /* check satisfiability of P_c over τ */
        if q_2 then
            if P_nf.child_2.type is ε then
                if |τ| == 0 then /* check whether the trace is empty */
                    return True;
                else
                    return False;
                end if
            else
                P_f = P_nf.child_2.child_1; /* obtain next formula P_f */
                if |τ| == 0 then
                    return False;
                else
                    τ = tail(τ); /* update τ by its first proper suffix */
                    return CheckBasedOnNF(P_f, τ);
                end if
            end if
        else
            return False;
        end if
end
```

Algorithm 4. function StateFormulaCheck(P_s, τ)

Input: P_s: A state PPTL formula;
Input: τ: A program execution trace;
Output: *True* if τ satisfies P_s, *False* otherwise.

```
begin
switch Ps.type do
    case OrProp /* Ps ≡ P1 ∨ P2 */
        P1 = Ps.child1;
        P2 = Ps.child2;
        q1 is a boolean variable;
        q1 = StateFormulaCheck(P1, τ);
        if q1 then
            return True;
        else
            return StateFormulaCheck(P2, τ)
        end if
    case AndProp /* Ps ≡ P1 ∧ P2 */
        P1 = Ps.child1;
        P2 = Ps.child2;
        q2 is a boolean variable;
        q2 = StateFormulaCheck(P1, τ);
        if q2 then
            return StateFormulaCheck(P2, τ)
        else
            return False;
        end if
    case NegationProp /* Ps ≡ ¬P1 */
        P1 = Ps.child1;
        if StateFormulaCheck(P1, τ) then
            return False;
        else
            return True;
        end if
    case AtomicProp /* Ps ≡ p */
        if head(τ) satisfies Ps then
            return True;
        else
            return False;
        end if
    case TrueProp /* Ps ≡ true */
        return True;
    case FalseProp /* Ps ≡ false */
        return False;
end
```

5 Conclusion

This paper presents an approach to mining PPTL specification from program execution traces. A tool PPTLMiner has been developed to support the proposed approach. This allows us to mine full regular temporal rules represented by PPTL formulas from traces. However, a mined PPTL formula has to be checked over all traces so as to ensure its validity. This is not a easy job since there might be error traces involved.

In the future, we will investigate how to evaluate the mined properties so that desired properties can be found. Further, we will optimize PPTLMiner to improve its mining quality and efficiency.

References

1. The Daikon Invariant Detector. http://plse.cs.washington.edu/daikon/
2. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar. IEEE Trans. Softw. Eng. **41**(7), 1 (2015)
3. Duan, Z.: Temporal logic and Temporal Logic Programming. Science Press, Beijing (2005)
4. Duan, Z., Tian, C.: A practical decision procedure for propositional projection temporal logic with infinite models. Theoret. Comput. Sci. **554**, 169–190 (2014)
5. Duan, Z., Tian, C., Zhang, L.: A decision procedure for propositional projection temporal logic with infinite models. Acta Informatica **45**(1), 43–78 (2008)
6. Duan, Z., Tian, C., Zhang, N.: A canonical form based decision procedure and model checking approach for propositional projection temporal logic. Theor. Comput. Sci. **609**, 544–560 (2016)
7. Duan, Z., Zhang, N., Koutny, M.: A complete proof system for propositional projection temporal logic. Theor. Comput. Sci. **497**, 84–107 (2013)
8. Dupont, P., Lambeau, B., Damas, C., Lamsweerde, A.: The QSM algorithm and its application to software behavior model induction. Appl. Artif. Intell. **22**(1&2), 77–115 (2008)
9. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002), pp. 411–420 (1999)
10. Iegorov, O., Fischmeister, S.: Mining task precedence graphs from real-time embedded system traces. pp. 251–260 (2018)
11. Le, T.B., Lo, D.: Deep specification mining. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 106–117 (2018)
12. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining (T). In: Proceedings of the 2015 IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 81–92 (2015)
13. Li, H., Shen, L.M., Ma, C., Liu, M.Y.: Role behavior detection method of privilege escalation attacks for android applications. Int. J. Perform. Eng. **15**(6), 1631–1641 (2019)
14. Narayan, A., Cutulenco, G., Joshi, Y., Fischmeister, S.: Mining timed regular specifications from system traces. ACM Trans. Embed. Comput. Syst. **17**(2), 1–21 (2018)

15. Pradel, M., Gross, T.R.: Automatic generation of object usage specifications from large method traces. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 371–382 (2009)
16. Ratcliff, S., White, D., Clark, J.: Searching for invariants using genetic programming and mutation testing. In: Proceedings of the 2011 Annual Genetic and Evolutionary Computation Conference, pp. 1907–1914 (2011)
17. Reger, G., Havelund, K.: What is a trace? A runtime verification perspective. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 339–355. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_25
18. Tian, C., Duan, Z.: Expressiveness of propositional projection temporal logic with star. Theor. Comput. Sci. **412**(18), 1729–1744 (2011)
19. Walkinshaw, N., Bogdanov, K., Holcombe, M., Salahuddin, S.: Reverse engineering state machines by interactive grammar inference. In: Proceedings of the 2007 Working Conference on Reverse Engineering, pp. 209–218 (2007)
20. Wasylkowski, A., Zeller, A.: Mining temporal specifications from object usage. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 295–306 (2009)
21. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: Proceedings of the 2006 International Conference on Software Engineering, pp. 282–291 (2006)