Satisfiability of Propositional Projection Temporal Logic *

Zhenhua Duan and Li Zhang

Institute of Computing Theory and Technology
Xidian University, Xi’an 710071 P.R. China
Email: zhhduan@mail.xidian.edu.cn

Abstract. This paper investigates the decision procedure for checking the satisfiability of Propositional Projection Temporal Logic (PPTL). The syntax, semantics and logic laws of PPTL are presented. Within this logic, besides the usual logic connectives, two basic temporal operators, $\circ$ and $prj$ are introduced. A normal form of PPTL is formalized and a proof for transforming a formula of PPTL into the normal form is given by induction on the structure of formulas in details. An algorithm transforming a formula into the normal form is presented. To find out a model for a given formula, the notation of Normal Form Graph (NFG) and an algorithm constructing NFG for the formula are introduced. A decision procedure for checking the satisfiability of PPTL formulas with finite models is demonstrated and some examples are given to illustrate how the algorithms work.

1 Introduction

Temporal logic [1–3] has been used to verify properties of programs for many years. Projection Temporal Logic (PTL) [4, 5] is an extension of Interval Temporal Logic (ITL) [6]. It introduces a new projection operator [7], $(P_1, \ldots, P_m) prj Q$, which can be thought of as a combination of the standard parallel and projection operators in ITL [6]. Intuitively, it means that $Q$ is executed in parallel with $P_1; \ldots; P_m$ over an interval obtained by taking the endpoints (rendezvous points) of the intervals over which $P_1, \ldots, P_m$ are executed.

The motivation for introducing the new projection construct is three folds: (1) to be able to handle terminal formulas (see Section 2); the original projection $(P proj Q)$ construct defined in [6] cannot deal with terminal formulas. With this definition, the formula $P$ is executed repeatedly over a series of consecutive subintervals whose endpoints form the interval over which $Q$ is executed. This may result in repeating the same global state in the execution of $Q$ several times if some of the copies of $P$ are executed over subintervals of zero length. Bowman and Thompson [8] changed this by introducing a similar construct in which $P$ is not permitted to be a terminable formula. However, this restrict causes the lose of flexibility since formula $P$ needs executed necessarily in a point state in

* This research is supported by the NSFC under Grant No. 60373103 and 60433010, and the SRFDP under Grant 20030701015.
some circumstances. In contrast, our definition rules this problem out; (2) to make formulas $P$ and $Q$ autonomous and terminate freely; in $P \text{ proj } Q$, the series of $P$’s and $Q$ always terminate at the same state. In practice, the formulas $P$’s and $Q$ are not so regular to be executed. Our projection construct permits the processes $P_1, \ldots, P_m$ and $Q$ to be autonomous; each process has the right to specify the interval over which it is executed. In particular, the sequence of processes $P_1, \ldots, P_m$ and process $Q$ may terminate at different time points. Although the communication between processes is still based on shared variables, the communication and synchronization only take place at the rendezvous points (global states), otherwise they are executed independently; (3) to make projection construct extensible to infinite intervals; both the original and Bowman’s projection constructs are defined over only finite intervals and it is very difficult to extend them to infinite intervals. In contrast, our projection construct is defined over both finite and infinite intervals.

Furthermore, the new projection construct subsumes the chop(;) $(P; Q \equiv (P, Q) \text{ prj empty})$ which is the central operator in ITL. Moreover, in the first order case, both the original and Bowman’s projection constructs can be defined using the new projection construct [9]. However, in the propositional logic, the original and the new projection operators are not directly comparable.

Within Propositional Projection Temporal Logic (PPTL), for instance, let $Pr$ be a finite set of atomic propositions and $p_1, p_2, p_3, q \in Pr$. Let $S = Pr\{p_1, p_2, p_3, q\}$ and $S_\lambda$ denote the conjunction of all of propositions in $S$. Thus, $(p_1, p_2, p_3) \text{ prj } q \wedge \Box S_\lambda$ is impossible to be expressed in Propositional Interval Temporal Logic (PITL). In this case, the lengths of models of formulas $p_1, p_2, p_3, q$ are all un-deterministic in PPTL, and processes $p_1; p_2; p_3$ and $q$ may terminate at different time points. $\Box S_\lambda$ specifies all propositions except those appearing in $\{p_1, p_2, p_3, q\}$ to be true in every state over the interval satisfying the formula $(p_1, p_2, p_3) \text{ prj } q$. Therefore, there is no way to express the formula within PITL.

We feel that although $P \text{ proj } Q$ and $(P_1, \ldots, P_m) \text{ prj } Q$ do share some important properties, they still possess sufficiently distinct features to be treated independently as complementary constructs useful in the programming environment in which different time scales need considered.

An interesting possibility for applying projection construct $(P_1, \ldots, P_m) \text{ prj } Q$ is that of hybrid systems. More specifically, one could treat $P_1, \ldots, P_m$ as formulas over dense intervals, and $Q$ as a formula that holds over a discrete interval consisting of the endpoints of the dense intervals [10, 11].

Satisfiability and validity of formulas are fundamental issues in the model theory of a logic. Furthermore, satisfiability plays an important role in the Model Checking (MC) approach which is an automatic approach based on MC algorithms and has been used for verification of programs for a long time. However, within ITL community, although several axiom systems have been proposed and verification of programs can be done in a deductive system [8, 12–16], Model Checking approaches have not extensively been studied. Within PTL, although plenty of logic laws have been formalized and proved [4, 7], a decision procedure for checking satisfiability of PPTL formulas is required since Model Checking is
closely relevant to satisfiability of the underlying logic. Therefore, it is necessary and important for us to investigate further satisfiability of PPTL formulas.

To this end, in the paper, a normal form of PPTL formulas is formalized. A proof for transforming formulas into the normal form is given in details. The proof proceeds by induction on the structure of formulas. An algorithm transforming a formula into the normal form is presented. To find out a model for a given formula, the notation of Normal Form Graph (NFG) and an algorithm constructing NFG for the formula are introduced. A decision procedure based on NFG for checking the satisfiability of PPTL formulas with finite models is demonstrated. With this method, for a given formula, an NFG can be constructed by means of its normal form. Thus, we do not need to consider different formulas separately since all formulas can be rewritten to this generic normal form. After the NFG is constructed, a reduction algorithm is used to remove redundant and inconsistent nodes. With the reduced NFG, a path from root node to the empty node corresponds to a model of the formula. Because we confine ourself to the finite models, the NFG of a formula is also finite (i.e. numbers of nodes and edges are finite). It is clear that a formula is satisfiable if and only if the empty node is in the set of nodes of its reduced NFG.

The paper is organized as follows. The following Section presents the syntax, semantics and logic laws of the underlying logic. In Section 3, the definition of the normal form of formulas is given; then a proof for reducing formulas to the normal form is presented; and an example to show how the algorithm works is provided. In section 4, the notation of Normal Form Graph (NFG) is presented and a decision algorithm for checking the satisfiability of PPTL formulas with finite models is also demonstrated. Conclusions are drawn in Section 5.

2 Propositional Projection Temporal Logic

Our underlying logic is a Propositional Temporal Logic with projection [4, 7]. It is an extension of Propositional Interval Temporal Logic [6].

Syntax Let Prop be the set of atomic propositions. A formula $P$ of PPTL is given by the following grammar:

$$P ::= p \mid \neg P \mid P_1 \lor P_2 \mid \square P \mid (P_1, \ldots, P_n) \text{prj} P$$

where $p \in \text{Prop}$. A formula is called a state formula if it does not contain any temporal operators (i.e. $\square$, $\text{prj}$) otherwise it is a temporal formula.

Semantics Following the definition of Kripke’s structure [17], we define a state $s$ over Prop to be a mapping from Prop to $B = \{\text{true, false}\}$: $s : \text{Prop} \rightarrow B$. We will use $s[p]$ to denote the valuation of $p$ at the state $s$. An interval $\sigma$ is a non-empty sequence of states, which can be finite or infinite. The length, $|\sigma|$, of $\sigma$ is $\omega$ if $\sigma$ is infinite, and the number of states minus 1 if $\sigma$ is finite. We consider the set $N_0$ of non-negative integers and $\omega$, $N_\omega = N_0 \cup \{\omega\}$ and extend the comparison operators, $=, <, \leq$, to $N_\omega$ by considering $\omega = \omega$, and for all $i \in N_0$, $i < \omega$. Furthermore, we define $\geq$ as $\leq - \{(\omega, \omega)\}$. To simplify definitions, we will denote $\sigma$ as $< s_0, \ldots, s_{|\sigma|} >$, where $s_{|\sigma|}$ is undefined if $\sigma$ is infinite. With such a notation, $\sigma_{(i..j)} (0 \leq i \leq j \leq |\sigma|)$ denotes the sub-interval $< s_i, \ldots, s_j >$. The
concatenation of a finite \( \sigma \) with another interval (or empty string) \( \sigma' \) is denoted by \( \sigma \cdot \sigma' \).

Let \( \sigma = < s_0, s_1, \ldots, s_{|\sigma|} > \) be an interval and \( r_1, \ldots, r_h \) be integers \((h \geq 1)\) such that \( 0 \leq r_1 \leq r_2 \leq \ldots \leq r_h \leq |\sigma| \). The projection of \( \sigma \) onto \( r_1, \ldots, r_h \) is the interval

\[
\sigma \downarrow (r_1, \ldots, r_h) = < s_{t_1}, s_{t_2}, \ldots, s_{t_l} >
\]

where \( t_1, \ldots, t_l \) are obtained from \( r_1, \ldots, r_h \) by deleting all duplicates. For instance,

\[
< s_0, s_1, s_2, s_3, s_4 > \downarrow (0, 0, 2, 2, 3) = < s_0, s_2, s_3 >
\]

So \((\text{empty}, \text{len}(2), \text{empty}, \text{empty}, \text{len}(1)) \) \( \text{proj} \) \( \text{len}(2) \) is in Fig.1.

![Fig. 1. The Projection of \( \sigma \)](image)

An interpretation is a quadruple \( I = (\sigma, i, k, j) \)\(^1\), where \( \sigma \) is an interval, \( i, k \) integers, and \( j \) an integer or \( \omega \) such that \( i \leq k \leq j \leq |\sigma| \). We use the notation \((\sigma, i, k, j) \models P \) to mean that some formula \( P \) is interpreted and satisfied over the subinterval \( < s_i, ..., s_j > \) of \( \sigma \) with the current state being \( s_k \). The satisfaction relation \( (\models) \) is inductively defined as follows:

\[
\begin{align*}
I &- \text{prop} \quad I \models p \text{ iff } s_k[p] = \text{true}, \text{ for an atomic proposition } p. \\
I &- \text{not} \quad I \models \neg P \text{ iff } I \not\models P. \\
I &- \text{or} \quad I \models P \lor Q \text{ iff } I \models P \text{ or } I \models Q. \\
I &- \text{next} \quad I \models \Box P \text{ iff } k < j \text{ and } (\sigma, i, k + 1, j) \models P. \\
I &- \text{proj} \quad I \models (P_1, \ldots, P_m) \text{ proj } Q \text{ iff there exist integers } k = r_0 \leq r_1 \leq \cdots \\
& \quad \leq r_m \leq j \text{ such that } (\sigma, i, r_0, r_1) \models P_1, (\sigma, r_{l-1}, r_{l-1}, r_l) \models P_l \\
& \quad (1 < l \leq m) \text{ and } (\sigma', 0, 0, |\sigma'|) \models Q \text{ for one of the following } \sigma'': \\
& \quad (a) \text{ } r_m < j \text{ and } \sigma'' = \sigma \downarrow (r_0, \ldots, r_m) \downarrow (r_{m+1}, j) \\
& \quad (b) \text{ } r_m = j \text{ and } \sigma'' = \sigma \downarrow (r_0, \ldots, r_h) \text{ for some } 0 \leq h \leq m.
\end{align*}
\]

Fig.2 shows the semantics of \((P_1, P_2) \text{ proj } Q\). (a) \( P_1; P_2 \) is finished before \( Q \); (b) \( Q \) and \( P_1; P_2 \) terminate at the same state; (c) \( Q \) terminates before \( P_2 \).

A formula \( P \) is satisfied by an interval \( \sigma \), denoted by \( \sigma \models P \), if \( (\sigma, 0, 0, |\sigma|) \models P \). A formula \( P \) is called satisfiable if \( \sigma \models P \) for some \( \sigma \).

**Derived formulas** The abbreviations \( \text{true}, \text{false}, \lor, \neg, \text{and} \leftrightarrow \) are defined as usual. In particular, \( \text{true} \overset{\text{def}}{=} P \lor \neg P \) and \( \text{false} \overset{\text{def}}{=} P \land \neg P \) for any formula \( P \). Furthermore, we use the following abbreviations:

---

\(^1\) The parameter \( i \) does not function here since it is used to handle past operators. However, to have a unified notation, it is kept in the interpretation.
Fig. 2. Semantics of \((P_1, P_2) \mathsf{prj} Q\)

\[
\begin{align*}
A1 & \text{ empty } \overset{\text{def}}{=} \neg \bigcirc \text{ true} & A2 & \text{ more } \overset{\text{def}}{=} \neg \text{ empty} \\
A3 & \bigcirc^0 P \overset{\text{def}}{=} P & A4 & \bigcirc^n P \overset{\text{def}}{=} \bigcirc(\bigcirc^{n-1} P) \\
A5 & \bigcirc P \overset{\text{def}}{=} \text{ empty } \lor \bigcirc P & A6 & \lozenge P \overset{\text{def}}{=} \text{ true } ; P \\
A7 & \Box P \overset{\text{def}}{=} \neg \Box \neg P & A8 & P ; Q \overset{\text{def}}{=} (P, Q) \text{ prj empty} \\
A9 & \text{ len}(n) \overset{\text{def}}{=} \bigcirc^n \text{ empty} & A10 & \text{ skip } \overset{\text{def}}{=} \text{ len}(1)
\end{align*}
\]

We denote \(\models \Box (P \iff Q)\) by \(P \equiv Q\). A formula \(P\) is called a terminal formula if \(P \equiv P \land \text{ empty}\). To avoid an excessive number of parentheses, the following precedence rules are used, where 1=highest and 5=lowest.

\[
1 \sim 2 \bigcirc, \bigcirc, \lozenge, \Box, 3 \land, \lor 4 \rightarrow, \iff 5 \text{ prj, ;}
\]

**Logic laws** Let \(P, Q, P_1, \ldots, P_m\) and \(R\) be PPTL formulas and \(w\) a state formula. Then the following logic laws hold.

\[
\begin{align*}
\text{FD1} & \quad \Box (P \land Q) \equiv \Box P \land \Box Q & \text{FD2} & \quad \lozenge (P \lor Q) \equiv \lozenge P \lor \lozenge Q \\
\text{FD3} & \quad \Box (P \land Q) \equiv \Box P \land \Box Q & \text{FD4} & \quad \lozenge (P \lor Q) \equiv \lozenge P \lor \lozenge Q \\
\text{FD5} & \quad \Box (P \land Q) \equiv \Box P \land \Box Q & \text{FD6} & \quad \lozenge (P \lor Q) \equiv \lozenge P \lor \lozenge Q \\
\text{FD7} & \quad (R; P \lor Q) \equiv (R; P) \lor (R; Q) & \text{FD8} & \quad (P \lor Q; R) \equiv (P; R) \lor (Q; R) \\
\text{FE1} & \quad \lozenge P \equiv P \lor \Box P & \text{FE2} & \quad \Box P \equiv P \land \Box P \\
\text{FM2} & \quad \text{ more } \land \neg \Box P \equiv \text{ more } \land \Box \neg P & \text{FDU1} & \quad \neg \Box P \equiv \Box \neg P \\
\text{FDU2} & \quad \neg \Box P \equiv \Box \neg P & \text{FDU3} & \quad \neg P \equiv \Box \neg P \\
\text{FDU4} & \quad \neg \Box P \equiv \neg \Box P & \text{FCH1} & \quad \Box P; Q \equiv \Box (P; Q) \\
\text{FCH2} & \quad w \land (P; Q) \equiv w \land P; Q & \text{FEP2} & \quad \text{ empty; } P \equiv P \\
\text{FEP5} & \quad Q; \text{ empty } \equiv Q \land \Box \text{ empty}
\end{align*}
\]

\[
\begin{align*}
\text{PRJe1} \quad (\text{ empty prj } Q) & \equiv Q & \text{PRJe2} \quad (Q \text{ prj empty}) & \equiv Q \\
\text{PRJe3} \quad (P_1, \ldots, P_m) \text{ prj empty } & \equiv P_1; \ldots; P_m \\
\text{PRJe4} \quad (P_1, \ldots, w \land P_{i+1}, \ldots, P_m) \text{ prj } Q & \equiv (P_1, \ldots, w \land P_{i+1}, \ldots, P_m) \text{ prj } Q \\
\text{PRJr1} \quad (P_1, \ldots, P_m) \text{ prj } Q & \equiv (P_1 \land \text{ more } ; ((P_2, \ldots, P_m) \text{ prj } Q)) \\
& \lor (P_1 \land \text{ empty } ; ((P_2, \ldots, P_m) \text{ prj } Q)) \\
\text{PRJr2} \quad (P_1, \ldots, P_m) \text{ prj } (P \lor Q) & \equiv ((P_1, \ldots, P_m) \text{ prj } P) \lor ((P_1, \ldots, P_m) \text{ prj } Q) \\
\text{PRJand1} \quad (w \land P_1, \ldots, P_m) \text{ prj } Q & \equiv w \land ((P_1, \ldots, P_m) \text{ prj } Q) \\
\text{PRJand2} \quad (P_1, \ldots, P_m) \text{ prj } (w \land Q) & \equiv w \land ((P_1, \ldots, P_m) \text{ prj } Q)
\end{align*}
\]

The proofs of these logic laws can be found in [4].
3 Normal Forms in PPTL

In this section, we present a normal form (NF) for the purpose of a decision procedure for checking the satisfiability of PPTL formulas.

Recall that in classical propositional logic a basic product (resp. sum) is a conjunction (resp. disjunction) of literals, where a literal is either $p$ or $\neg p$ for some atomic proposition $p$. A state formula is said to be in disjunctive (resp. conjunctive) normal form (DNF) (resp. CNF) if it is the sum, ‘or’ in logic (resp. product, ‘and’ in logic) of products of literals. A basic product (resp. sum) is called min-product (resp. max-sum) if, for all atomic propositions, either a proposition or its negation but not both appears in the product (resp. sum). If there are $n$ atomic propositions then we have $2^n$ min-products, $m_0, m_1, ..., m_{2^n-1}$ and $2^n$ max-sums, $M_0, M_1, ..., M_{2^n-1}$. The conjunction of any two different min-products is $false$ and the disjunction of all min-products is $true$ (in this sense, we say that the set of all min-products is complete). That is,

\[ \bigwedge_{i} m_i = true \]  
\[ \bigvee_{i} (m_i \land m_j) = false \]  

Thus, for any formula $Q_i$, it is not difficult to see the following property.

\[ \neg \bigwedge_{i} (m_i \land Q_i) \equiv \bigvee_{i} (m_i \land \neg Q_i) \]  

This property can be easily generalized to a set of formulas $< P_i >_i$. We call $< P_i >_i$ complete if $< P_i >_i$ satisfies the following properties:

\[ \bigwedge_{i} P_i \equiv true \]  
\[ \bigvee_{i \neq j} (P_i \land P_j) \equiv false \]  

In this case, it is easily to show that (3.3) is still valid, i.e.

\[ \neg \bigwedge_{i} (P_i \land Q_i) \equiv \bigvee_{i} (P_i \land \neg Q_i) \]  

This idea has been used by Kono [18], Bowman and Thompson [8]. For convenience, let $L_{pptl}$ denote the set of all PPTL formulas, and $Q_p \subseteq Prop$ be the set of all atomic propositions in formula $Q$. We now define the normal form of PPTL formulas as follows.

**Definition 3.1** Let $Q \in L_{pptl}$. $Q$ is in a normal form if $Q$ has been written as $Q \equiv \bigvee_{i=1}^n ((\bigwedge_{j=1}^m \hat{q}_{ij}) \land \bigodot Q'_i)$  
where \( l = |Q_p|, 1 \leq n \leq 3^l, 1 \leq m \leq l \), $q_{ij} \in Q_p$ and $\hat{q}_{ij}$ denotes $q_{ij}$ or $\neg q_{ij}$ and $Q'_i \in L_{pptl}$ is a general formula.

Notice that the following points are important about the normal form. (1) We need to explain the number $3^l$. Because $\bigwedge_{j=1}^m \hat{q}_{ij}$ is a basic product, an atomic proposition and its negation do not appear in it simultaneously. So, for any proposition $p \in Q_p$, $p$ or $\neg p$ or neither can appear in the basic product. Furthermore, the number of atomic propositions in formula $Q$ is $l$. Therefore, the number of all basic products is at most $3^l$. (2) The normal form can be written in various forms in different circumstances. To separate the empty part and future part explicitly, we write the normal form as

\[ Q \equiv \bigvee_{i=1}^{m_0} (\bigwedge_{j=1}^m \hat{q}_{ij}) \land empty \lor \bigvee_{i=1}^{m_1} (\bigwedge_{j=1}^m \hat{q}_{ij}) \land \bigodot Q'_i \]  

For simplicity, sometimes we write the normal form as

\[ Q \equiv \bigvee_{i=1}^n (w_i \land \bigodot Q'_i) \]  

or
\[ Q \equiv \bigvee_{j=1}^{n} (w_{ej} \land \text{empty}) \lor \bigvee_{i=1}^{n} (w_{i} \land Q_{i}') \]  

(3.8)

where \( w_{ej} \) and \( w_{i} \) are state formulas. Sometimes, we also write \( w_{e} \land \text{empty} \) instead of \( \bigvee_{j=1}^{n} w_{ej} \land \text{empty} \). (3) A normal form is called complete when \( w_{i} >_{i} \) is complete.

**Lemma 3.1** If a formula \( P \) has been rewritten to its normal form, then \( P \) can also be rewritten to its complete normal form.

**Proof** Suppose \( P \) has been rewritten to its normal form, \( P \equiv P_{e} \land \text{empty} \lor \bigvee_{i=1}^{n} (P_{i} \land Q_{i}') \) where \( P_{e} \) and \( P_{i} \) are state formulas.

We can treat \( P_{i} \) as atoms and construct \( 2^{n} \) min-products, \( m_{0}, \ldots, m_{2^{n}-1} \), whereas we can treat \( P_{i}' \) as atoms and construct \( 2^{n} \) max-sums \( M_{0}, \ldots, M_{2^{n}-1} \).

From these max-sums, we can obtain \( 2^{n} \) basic sums namely \( M_{0}', \ldots, M_{2^{n}-1}' \), here \( M_{j}' \) denotes \( M_{2^{n}-j} \) by deleting negative disjuncts and \( M_{2^{n}-1}' \) denotes \( \text{false} \).

Thus, it is readily to prove the following:

\[ P \equiv P_{e} \land \text{empty} \lor \bigvee_{i=1}^{n} (m_{i} \land \bigvee_{j=1}^{n} (M_{j}') \lor \text{false}) \]

(3.9)

Obviously, \( P \) is now in the complete normal form. \( \square \)

For instance,

\[ (p \land \bigvee Q') \lor (q \land \bigvee Q') \equiv ((p \land q) \lor (p \land q) \lor (p \land q)) \]

\[ \lor ((\neg p \land q) \lor (\neg p \land q) \lor (\neg p \land q)) \]

Lemma 3.1 is useful for transforming a formula in the negation form into its normal form while Lemma 3.2 is concerned with transforming the projection construct into its normal form.

**Lemma 3.2** Let \( R \equiv (P_{1}, \ldots, P_{m}) \land Q \). Suppose \( P_{1}, \ldots, P_{m} \) and \( Q \) have been rewritten to the normal form, then \( R \) can be rewritten to its normal form.

**Proof** The proof proceeds by induction on \( m \).

Suppose \( P_{1} \equiv P_{e} \land \text{empty} \lor \bigvee_{i=1}^{n} (P_{i} \land Q_{i}') \)

\( Q \equiv Q_{e} \land \text{empty} \lor \bigvee_{i=1}^{n} (Q_{i} \land Q_{i}') \)

where \( P_{e}, Q_{e}, P_{i}, Q_{i} \) are state formulas.

\( (P_{1}) \land Q \equiv P_{1} \land Q_{e} \land \text{empty} \)

\[ \lor \bigvee_{i=1}^{n} (P_{i} \land Q_{i} \land \bigvee_{j=1}^{n} (P_{j}') \lor \text{false}) \]

\[ \lor \bigvee_{i=1}^{n} (P_{i} \land Q_{i} \land \bigvee_{j=1}^{n} (P_{j}') \lor \text{false}) \]

where \( P_{1e} \land Q_{e}, Q_{1}, P_{1i}, Q_{1i} \) are state formulas.

Suppose \( (P_{2}, \ldots, P_{m}) \land Q \equiv R_{e} \land \text{empty} \lor \bigvee_{j=1}^{n} (R_{j} \land R_{j}') \) where \( R_{e} \) and \( R_{j} \) are state formulas. Thus,

\( (P_{1}, \ldots, P_{m}) \land Q \equiv P_{e} \land R_{e} \land \text{empty} \)

\[ \lor \bigvee_{i=1}^{n} (P_{i} \land R_{i} \land \bigvee_{j=1}^{n} (R_{j}')) \]

\[ \lor \bigvee_{i=1}^{n} (P_{i} \land Q_{i} \land \bigvee_{j=1}^{n} (P_{j}')) \]

\[ \lor \bigvee_{i=1}^{n} (P_{i} \land Q_{i} \land \bigvee_{j=1}^{n} (P_{j}')) \]

where \( P_{1e} \land R_{e}, R_{1}, P_{1i}, Q_{1i} \) and \( P_{1i} \land Q_{i} \) are state formulas. \( \square \)

**Theorem 3.3** For any PPTL formula \( R \), \( R \) can be rewritten to its normal form defined in Definition 3.1.
Proof The proof proceeds by induction on the structure of formulas. Suppose,
\[ P \equiv P_r \land \emptyset \lor \bigvee_{i=1}^n (P_i \land \bigcap P_i') \]
\[ Q \equiv Q_r \land \emptyset \lor \bigvee_{k=1}^m (Q_k \land \bigcap Q_k') \]
where \( P_r, Q_r, P_i, Q_k \) are state formulas.

**Base:** If \( R \equiv p \) and \( p \in R_p \), then we have \( R \equiv p \land (\emptyset \lor \text{more}) \equiv p \land \top \).

**Induction:**

1. If \( R \equiv P \land Q \), then the proof is straightforward.
2. If \( R \equiv P \lor Q \), then we have \( R \equiv \top \land \bigvee_{i=0}^{2h-1} m_i \land \bigcap P \) where \( m_i \) is a min-product consisting of atomic propositions in \( R_p \) and \( h = |R_p| \).
3. If \( R \equiv (P_1, \ldots, P_m) \) \( prj Q \) then, by Lemma 3.2, the conclusion is true.
4. If the formula \( R \) is in the form \( \neg P \), by Lemma 3.1, we assume \( P \) is in the complete normal form. Thus,
\[ \neg P \equiv \neg P_{\emptyset} \land \emptyset \lor \bigvee_{i=1}^n (P_i \land \neg P_i') \]
where \( \neg P_{\emptyset} \) and \( P_i \) are state formulas. \( \square \)

We now give an algorithm \( NF \) for transforming a PPTL formula into its normal form. This algorithm is valid for any PPTL formula regardless of finite or infinite models. The basic idea of the algorithm is the same as the above proof. Accordingly, the above theorem and lemmas ensure the correctness of algorithm \( NF \). Given a formula \( R \) as input, we first do some preprocesses by \( PRE(R) \) to eliminate all implications, equivalences, double negations in \( R \) by replacing all sub-formulas of the form \( P \rightarrow Q \) by \( \neg P \lor Q \), \( P \leftrightarrow Q \) by \( \neg P \land Q \lor P \land \neg Q \) and \( \neg \neg P \) by \( P \). However, we cannot get rid of all negations since the negation of \( prj \) construct (resp. \( \text{chop}() \)) is not eliminable. Finally, we do some postprocesses by \( POST(R) \) to simplify \( R \) by replacing all products of the form \( w \land \bigcap P \lor w \land \bigcap Q \) by \( w \land \bigcap (P \lor Q) \). Algorithm \( DNF \) is to translate a state formula into its disjunctive normal form. And algorithm \( NF DNF \) is to transform a disjunctive normal form into our normal form by replacing all basic products \( \bigwedge_{j=1}^n q_{ij} \) by \( \bigwedge_{j=1}^n \neg q_{ij} \land \emptyset \lor \bigvee_{j=1}^n q_{ij} \lor \top \). We have omitted codes for \( DNF \), \( PRE \) and \( POST \) here. We say that a formula \( P \) is in \( PRE \) (resp. \( POST \), \( NF \), etc.) if \( P \) has been processed by \( PRE(P) \) (resp. \( POST(P) \), \( NF(P) \), etc.). In the following algorithms, \( w \) denotes a state formula and \( \mu \) a basic product. We can now write pseudo codes for \( NF \) as follows:

```plaintext
function NF(R)
/* precondition: \( R \in \text{L_pptl} \) and in \( PRE\)*/
/* postcondition: \( NF(R) \) computes an equivalent NF for \( R \) */
begin function
    case
        R is true: return empty \lor \top
        R is false: return p \land \neg p \lor \emptyset \lor \neg p \land \top
        R is a state formula: return NF DNF (DNF(R))
        R is \( \mu \land \top \): return R
        R is \( \mu \land \emptyset \): return R
        R is \( P \lor Q \): return NF(P) \lor NF(Q)
end function
```
$R$ is $(P_1, \ldots, P_m) \ prj Q$: return $PRJ(R)$

$R$ is $\neg P$: return $NEG(CONF(NF(P)))$

$R$ is $\bigcirc P$: return $NF(P) \lor \bigcirc P$

$R$ is $\bigoplus P$: return $NF(P \land empty) \lor NF(P \land \bigcirc \bigoplus P)$

$R$ is $P \land Q$: return $AND(NF(P), NF(Q))$

end function

end function

In Algorithm $NF(R)$, the basic constructs such as atomic propositions, the next, disjunction, negation and projection constructs need considered. However, to improve the efficiency, other constructs such as sometimes, always, and chop are also taken into account. Intuitively, if $R$ is $true$, $false$ or a state formula, the transformation is straightforward; if $R$ is $\mu \land \bigcirc P$ or $\mu \land empty$, $R$ is already in its normal form; if $R$ is $P \lor Q$ the algorithm only needs to transform formulas $P$ and $Q$ into the normal form; if $R$ is $(P_1, \ldots, P_m) \ prj Q$ the algorithm calls function $PRJ$ to rewrite it; in addition, $\bigcirc P \equiv P \lor \bigcirc \bigoplus P$, $\bigoplus P \equiv P \lor \mu \land \bigcirc \bigoplus P$, and $\bigcirc \bigoplus P$ is in the normal form, so the algorithm needs to transform $P$, $P \land empty$, $P \lor \bigcirc \bigoplus P$ into the normal form; for $P \land Q$, the algorithm transforms $P$ and $Q$ into the normal form at first, then rewrites $P \land Q$ into its normal form with distributivity laws.

function $PRJ(R)$

/* precondition: $R \equiv (P_1, \ldots, P_m) \ prj Q \in L_{pat}$ and in $PRE$ */

/* postcondition: $PRJ(R)$ computes an equivalent NF for $R$ */

begin function

case

$R$ is $empty \ prj Q$: return $NF(Q)$

$R$ is $P \ prj empty$: return $NF(P)$

$R$ is $(\bigcirc P_1, \cdots, P_m) \ prj \bigcirc Q$: return $\bigcirc(P_1:(P_2, \cdots, P_m) \ prj Q))$

$R$ is $(\bigcirc P_1, \cdots, P_m) \ prj \bigoplus Q$: return $PRJ((P_2, \cdots, P_m) \ prj Q)$

$R$ is $(P_1, \cdots, P_m) \ prj empty$: return $CHOP(P_1:(P_2; \cdots; P_m))$

$R$ is $(w \land P_1, \cdots, P_m) \ prj Q$: return $NF(w \land PRJ((P_1, \ldots, P_m) \ prj Q))$

$R$ is $(\bigoplus P_1, \cdots, P_m) \ prj Q$: return $PRJ((P_1, \cdots, P_m) \ prj Q) \lor PRJ((P'_1, P_2, \cdots, P_m) \ prj Q)$

$R$ is $(P_1, \cdots, P_m) \ prj (Q_1 \lor Q_2)$:

otherwise: return $PRJ((NF(P_1), \cdots, P_m) \ prj NF(Q))$

end case

end function

Algorithm $PRJ$ is straightforward. It considers all possible projection constructs and uses the corresponding logic laws to rewrite them. These laws are $PRJ_{e1}$, $PRJ_{e2}$, $PRJ_{e3}$, $PRJ_{e4}$, $PRJ_{and1}$, $PRJ_{and2}$, $PRJ_{or1}$ and $PRJ_{or2}$. When the algorithm deals with $(P_1, \ldots, P_m) prj empty \equiv P_1; \ldots; P_m$, it calls function $CHOP$ to rewrite it. With algorithm $CHOP$, all possible cases are taken into account and the relative logic laws are employed to deal with them. These laws include $FEP2$, $FCH1$, $FCH2$, $FD9$ and $FD10$. 
function CHOP(R)
/* precondition: R ≡ P; Q ∈ L_{ppt} and in PRE */
/* postcondition: CHOP(R) computes an equivalent NF for R */
begin function
  case R is empty : Q: return NF(Q)
  R is ⊓P : Q: return ⊓(P ; Q)
  R is w ∧ P : Q: return NF(w ∧ CHOP(P ; Q))
  R is P₁ ∨ P₂ : Q: return CHOP(P₁ ; Q) ∨ CHOP(P₂ ; Q)
  R is P ; Q₁ ∨ Q₂: return CHOP(P ; Q₁) ∨ CHOP(P ; Q₂)
  otherwise: return CHOP(NF(P) ; NF(Q))
end case
end function

For formula ¬P, we first rewrite formula P into its normal form, then we convert it into its complete normal form by algorithm CONF which is simply based on (3.9). After that, by applying (3.4), ¬P can be rewritten to its normal form. The ideas are shown in algorithms CONF and NEG, respectively.

function CONF(R)
/* precondition: R is in the normal form, mᵢ and Mᵢ are defined as in Lemma 3.1 */
/* postcondition: CONF(R) computes an equivalent complete normal form for R */
begin function
  case R is P ∧ ⊓P’: return P ∧ ⊓P’ ∨ ¬P ∧ false
  R is P ∧ ⊓P’ ∨ ∨ᵢ=1(mᵢ ∧ ⊓Mᵢ):
      return ∨ᵢ=1(P ∧ mᵢ ∧ ⊓P’) ∨ ∨ᵢ=1(¬P ∧ mᵢ ∧ ⊓Mᵢ)
  R is Pₑ ∧ empty ∨ ∨ᵢ=1(Pᵢ ∧ ⊓Pᵢʼ):
      return Pₑ ∧ empty ∨ CONF(Pᵢ ∧ ⊓Pᵢʼ) ∨ CONF(∃ᵢ=1,Pᵢ ∧ ⊓Pᵢʼ))
end case
end function

function NEG(R)
/* precondition: R ≡ Pₑ ∧ empty ∨ ∨ᵢ=1(mᵢ ∧ ⊓Mᵢ) is in complete normal form */
/* postcondition: NEG(R) computes an equivalent NF for ¬R */
begin function
  return ¬Pₑ ∧ empty ∨ ∨ᵢ=1(mᵢ ∧ ¬Mᵢ)
end function

In algorithm AND, we assume that P and Q are in the normal form. So P(resp. Q) is in the form of P₁ ∨ P₂, Pₑ ∧ empty or Pₑ ∧ ⊓Pₑ’. If P is in the form of P₁ ∨ P₂, then algorithm AND deals with (P₁ ∧ Q) ∨ (P₂ ∧ Q) and transforms (P₁ ∧ Q) and (P₂ ∧ Q) by calling itself. If P ≡ Pₑ ∧ empty, Q ≡ Qₑ ∧ empty or P ≡ Pₑ ∧ ⊓Pₑ’, Q ≡ Qₑ ∧ ⊓Qₑ’ then the transforming is straightforward. If P ≡ Pₑ ∧ empty, Q ≡ Qₑ ∧ ⊓Qₑ’ or P ≡ Pₑ ∧ ⊓Pₑ’, Q ≡ Qₑ ∧ empty then P ∧ Q ≡ false and the algorithm needs only to call algorithm NF to transform
We now try to transform formula (end function
4 Normal Form Graph and Decision Algorithm
For convenience of descriptions in the following,
AND function into the normal form./* precondition: P and Q are in the normal form */
begin function
/* postcondition: AND(P, Q) computes an equivalent NF for P \land Q */
begin case
P is P_1 \lor P_2; return \text{AND}(P_1, Q) \lor \text{AND}(P_2, Q)
Q is Q_1 \lor Q_2; return \text{AND}(P, Q_1) \lor \text{AND}(P, Q_2)
P is P_e \land \text{empty} and Q is Q_e \land \text{empty}; return P_e \land Q_e \land \text{empty}
P is P_1 \land \bigcirc P'_1 and Q is Q_2 \land \bigcirc Q'_2; return P_1 \land Q_2 \land \bigcirc (P'_1 \land Q'_2)
P \land Q contains \text{empty} \lor \bigcirc R. return NF(\text{false})
end case
end function

Example 3.1 We now try to transform formula (P_1, P_2) \text{prj} Q into its normal form by means of the above algorithms, where P_1 \equiv \Box p, P_2 \equiv \Box(\text{empty} \lor q) \land \bigcirc \text{empty}, Q \equiv \text{empty} \lor q.
\text{POST}(\text{NF}([(P_1, P_2) \text{prj} Q]))
\equiv \text{POST}((P_1, P_2) \text{prj} Q))
\equiv \text{POST}(\text{PRJ}((P_1, P_2) \text{prj} \text{empty})) \lor \text{PRJ}((P_1, P_2) \text{prj} q))
\equiv \text{POST}(\text{CHOP}(P_1, P_2)) \lor \text{PRJ}((\text{NF}(P_1), P_2) \text{prj} \text{NF}(q)))
\equiv \text{POST}(\Box(p : P_2) \lor \text{PRJ}((\Box p, P_2) \text{prj} (q \land \text{empty} \lor q \land \bigcirc \text{true})))
\equiv \text{POST}(\Box(p : P_2) \lor \text{PRJ}((\Box p, P_2) \text{prj}(q \land \text{empty})))
\lor \text{PRJ}((\Box p, P_2) \text{prj} q \land \bigcirc \text{true}))
\equiv \text{POST}(\Box(p : P_2) \lor \Box(\text{NF}(q \land \text{CHOP}(\Box p : P_2)))
\lor \text{NF}(q \land \text{PRJ}((\Box p, P_2) \text{prj} \bigcirc \text{true})))
\equiv \text{POST}(\Box(p : P_2) \lor q \land \bigcirc (p : P_2) \lor q \land \bigcirc(p; (P_2 \text{prj} \text{true})))
\equiv \Box(p : P_2) \lor q \land \bigcirc((p : P_2) \lor (p : (P_2 \text{prj} \text{true})))

4 Normal Form Graph and Decision Algorithm
For convenience of descriptions in the following, w_ej \land \text{empty} is said to be a terminal product and w_i \land \bigcirc R'_i is known as a future product; w_ej and w_i are called present components while R'_i are called next components. The normal form of formula R can be presented by a set called normal form set R_s. Each member of R_s corresponds to a terminal or future product and is again presented in a set called component set consisting of sub-formulas which are the components within a terminal or future product. For instance, R \equiv p \land \text{empty} \lor p \land \bigcirc q can be denoted by R_s = \{[p, \text{empty}], [p, \bigcirc q]\} or R_s = \{S_1, S_2\} where S_1 = [p, \text{empty}] and S_2 = [p, \bigcirc q]. Clearly, members in the normal form set are disjunctive whereas members in each component set are conjunctive. So we also use S_i to denote a formula corresponding to a component set. For example, S_{1,2} = p \land \text{empty}, S_{2,3} = p \land \bigcirc q.

Armed with the normal form and its generating algorithm, a formula R can be decomposed to a so called Normal Form Graph (NFG) as follows: Initially,
the root (denoted by a small double circle) of the graph is labelled by formula $R$; each terminal or future product in the normal form of $R$ generates a son of $R$. With the terminal product, the edge is labelled by present component $R_{ej}$ and a terminal node (a small black dot) labelled by $\varepsilon$ without appearing of empty; and with the future product, the edge is labelled by $R_i$ and the next node (a small circle) labelled by next component $R_i'$. Then, $R_i'$ can further be decomposed to a sub-graph of $R$ and so on. If two nodes are identical, we merge them into one. It is clear that if $R$ has only finite models, its NFG is also finite.

For any formula $R$ in $L_{pptl}$, if $R$ is satisfiable, a model of $R$ can be finite or infinite; the number of models of $R$ can also be finitely or infinitely many. To work out a decision procedure for checking the satisfiability of PPTL formulas, we confine ourselves to finite models. That is, we consider only formulas with finite models. In the following, an algorithm called NFGA is given to generate the NFG of formula $R$. In the algorithm, global variables $V$ and $E$ denote the set of nodes and the set of edges, respectively. An edge from node $R$ to node $R_i'$ indicates that $R_i'$ is a temporal successor of $R$. We refer to such an edge as $< R, R_{ej}, \varepsilon >$ or $< R, R_i, R_i' >$ depending on it is a terminal product or future product. Algorithm REMOVE($G$) eliminates redundant nodes, i.e., nodes labeled with $F$ or without successors, where $G = (V, E)$ is an NFG produced by algorithm NFGA. Pseudo codes for NFGA and REMOVE can be given as follows.

```plaintext
function NFGA($R, V, E$)
/* precondition: $R \in L_{pptl}$ in NF and POST, $G = (V, E)$, $V = \{R\}$, $E = \{\}$ */
/* postcondition: NFGA($R, V, E$) computes an equivalent NFG for $R$ */
begin function
if $R \equiv \bigvee_{j=1}^m (R_{ej} \land empty) \lor \bigvee_{i=1}^n (R_i \land \cdots R_i')$ /*Remove all terminal or future products which equal false*/
for each element $S$ in $R_s$ do /*$R_s$ is a normal form set.*/
if $p \in S$ and $\neg p \in S$ then remove $S$ from $R_s$
end for
/*Label the unsatisfiable node with $F$*/
if $R_s == \phi$ then the node is labelled by $F$ and return $G$
end if
/*Produce new nodes and edges.*/
for each element $S$ in $R_s$ do
/*$S_\wedge$ is a formula corresponding to a component set $S^*$*/
/*Deal with a terminal product*/
if $S_\wedge = R_{ej} \land empty$ then $V := V \cup \{\varepsilon\}$
$E := E \cup \{< R, R_{ej}, \varepsilon >\}$
end if
/*Deal with a future product*/
if $S_\wedge = R_i \land \cdots R_i'$ then $V := V \cup \{R_i'\}$
$E := E \cup \{< R, R_i, R_i' >\}$
NFGA(POST(NF(PRE($R_i'$)))$), V, E)$
end if
end if
```

```plaintext
function REMOVE($G$)
/* precondition: $G = (V, E)$ is an NFG */
/* postcondition: REMOVE($G$) eliminates redundant nodes, i.e., nodes labeled with $F$ or without successors, where $G = (V, E)$ is an NFG */
begin function
for each element $S$ in $R_s$ do
if $p \in S$ and $\neg p \in S$ then remove $S$ from $R_s$
end for
/*Label the unsatisfiable node with $F$*/
if $R_s == \phi$ then the node is labelled by $F$ and return $G$
end if
/*Produce new nodes and edges.*/
for each element $S$ in $R_s$ do
/*$S_\wedge$ is a formula corresponding to a component set $S^*$*/
/*Deal with a terminal product*/
if $S_\wedge = R_{ej} \land empty$ then $V := V \cup \{\varepsilon\}$
$E := E \cup \{< R, R_{ej}, \varepsilon >\}$
end if
/*Deal with a future product*/
if $S_\wedge = R_i \land \cdots R_i'$ then $V := V \cup \{R_i'\}$
$E := E \cup \{< R, R_i, R_i' >\}$
NFGA(POST(NF(PRE($R_i'$)))$), V, E)$
end if
end if
```
```
function REMOVE(G)

/* precondition: NFG G = (V,E)*/
/* postcondition: REMOVE(G) obtains NFG G without redundant nodes */
begin function
for each node R ∈ V do
    if R is labelled by F or has no successors
        then V = V\{R}
        for each < P,w,R >∈ E do
            E = E\{< P,w,R >}
        end for
    end if
end for
return G
end function

For convenience, we give each node of NFG an identifier v_i (1 ≤ i ≤ |V|) besides its labeling and the root has an identifier v_1. A path is an alternant sequence of nodes and edges and corresponds to a model. Algorithm NFGA(R, V, E) terminates as long as R only has finite models. In this case, the NFG of P has a terminal node. Therefore, if node ε is not in V, P is not satisfiable, otherwise it is satisfiable. We can trace back from node ε to the root within NFG of R to find a model (a path) satisfying formula R. For instance, Fig.3 depicts an NFG of the formula in Example 3.1. Formulas in the nodes are placed in the right column of the figure. The path < v_1, q, v_3, p∧q, v_6, true, ε > determines a model σ =< {q}, {p,q}, {true} >, where, true stands for any propositions.

**Theorem 4.1** Let P be a formula and G = (V,E) be NFG of P. P is satisfiable if and only if ε ∈ V.

Finally, we have the following decision algorithm for given formula R:
function DECISION(R)
/* precondition: R ∈ L_{ppl} and in NF and POST */
/* postcondition: DECISION(R) determines the satisfiability of R */
begin function
    G = REMOVE(NFGA(R))
    if ε ∈ V then return true else return false
end if
end function

**Discussion** The algorithm we presented here is simpler in a number of respects than that defined in [8] although both algorithms are based on the normal form mechanism. For example, given a formula P, the NFG of P is much smaller than the Tableau graph generated by the rules in [8]. With the latter, the number of nodes in the graph is the sum of the number of nodes generated by decomposition
and step rules; while, with the former, the construction process is based on the
decomposition of \( P \) into its normal form. In contrast with the composition and
step processes in [8], this approach is in one go: so the number of nodes in the
NFG of \( P \) is roughly the half of that in the Tableau graph. Furthermore, with
the latter, a 'guard' \( \mathsf{empty} \) is required be conjunctive with formula \( P \) so that
unaccepted formulas can be recognized, while with the former, this guard is not
needed. Moreover, \( P \) is satisfiable as long as, with the latter, \( P \) is in the initial
node in the reduced Tableau graph while with the former \( \varepsilon \) is in the reduced
NFG of \( P \). However, a model of \( P \) in the NFG is a path from root to the \( \varepsilon \)
node while in the Tableau graph, a model of \( P \) is not obvious. For instance,
the Tableau graph of formula \( \Box \Diamond q \) is of 12 nodes and 12 edges as shown in
Figure 3 in [8] while the NFG of the formula has only 5 nodes and 6 edges as
shown in Fig.4. A model, \(< \{ \text{true} \}, \{ \text{true} \}, \{ q \} > \), of the formula can easily be
constructed from path \(< v_1, \text{true}, v_2, \text{true}, v_3, q, \varepsilon > \).

Fig. 3. The Normal Form Graph

\[
\begin{align*}
v_1 : (P_1, P_2) & \quad prj \ Q \\
v_2 : & \quad p \ ; \ P_2 \\
v_3 : (p \ ; \ P_2) & \quad \lor \ (p \ ; \ (P_2 \ prj \ true)) \\
v_4 : & \quad \Box(\emptyset \lor q) \land \ empty \\
v_5 : & \quad true \ ; \ P_2 \\
v_6 : & \quad \Box(\emptyset \lor q) \land \ empty \lor (\Box(\emptyset \lor q) \land \ empty \land true) \\
v_7 : & \quad (true \ ; \ P_2) \lor (true \ ; \ (P_2 \ prj \ true)) \\
v_8 : & \quad true
\end{align*}
\]

Fig. 4. The NFG of \( \Box \Diamond q \)
5 Conclusion

The PPTL presented in this paper is intended for the verification of properties of programs based on Model Checking. The definition and proof of the normal form as well as the algorithm transforming a formula into the normal form can be used to any formulas regardless of its models being finite or infinite. The decision procedure for checking the satisfiability of PPTL formulas can only be used for a formula with finite models. Therefore, MC algorithm with PPTL is limited to finite model checking. In the future, we will develop MC algorithms based on PPTL to verify properties of programs.

References

Appendix

Proof of Theorem 4.1

(⇒) If P is satisfiable then ε ∈ V.
Suppose there is model σ, σ |= P ⇔ j=1(Pc ∧ empty) ∨ ∃i=1(Pi ∧ NFGA). We
try to prove ε ∈ V. The proof proceeds by induction on the length of σ.
Base: If |σ| = 0, then ∃j, 1 ≤ j ≤ l, such that σ |= Pcj. By algorithms NFGA and
REMOVE, v1 ∈ V, labelled by P, edge < P, Pcj, ε > ∈ E, hence ε ∈ V.
Induction: Suppose for all σ, |σ| = k − 1 (k ≥ 1), the conclusion is true. If
|σ| = k ≥ 1, then ∃i, 1 ≤ i ≤ n, such that σ |= Pi ∧ NFGA, leading to σ |= Pi
and σ |= NFGA. This is, (σ, 0, 0, |σ|) = Pi and (σ, 0, 1, |σ|) = NFGA. By algorithms
NFGA and REMOVE, v1 ∈ V, labelled by Pi, 3v ∈ V, labelled by NFGA, and
< P, Pi, NFGA > ∈ E. Let σ′ = σ(1, |σ|), thus, (σ′, 0, 0, |σ′|) = NFGA, i.e σ′ |= Pi
and |σ′| = |σ| − 1 = k − 1. Let G′ = (V′, E′) be the NFG of σ′, v ∈ V′ and labelled
by NFGA. By inductive hypothesis, ε ∈ V′. It is clear that V′ ⊂ V and E′ ⊂ E, so
ε ∈ V.

(⇐) If ε ∈ V then P is satisfiable.
Since ε ∈ V, v1 ∈ V labelled by P, there is a path s = < v1, P1, v2, P2, ..., vk, Pk, ε >
in G. We need to prove that, for |s| ≥ 1, interval σ can be constructed so that
σ |= P. The proof proceeds by induction on the length of path s.
Base: If |s| = 1, i.e, k = 1, then P1 = Pcj and < P, Pcj, ε > ∈ G. This means that
Pc ∧ empty is a terminal product in the normal form of formula P. We can
construct interval σ = < s0 > such that for all p contained in Pcj s0[p] = true
and for all ¬p contained in Pcj s0[p] = false. Thus, obviously, σ |= P.
Induction: Suppose |s| = k − 1 (k ≥ 1) the conclusion is true. If |s| = k,
then s = < v1, P1, v2, P2, ..., vk, Pk, ε >, and let s1 = < v2, P2, ..., vk, Pk, ε >
and |s1| = k − 1. Suppose v2 is labelled by P1, by hypothesis, we can
construct a model σ′, σ′ |= P1. Then we can construct a new interval as follows.
σ = < s0 > · σ′, where s0 is a state such that for all proposition p contained in
P1, s0[p] = true and for all ¬p contained in P1 s0[p] = false, so < s0 > |= P1.
Thus, σ |= NFGA and σ |= P1, leading to σ |= Pi ∧ NFGA.

P ⇔ j=1(Pc ∧ empty) ∨ ∃i=1(Pi ∧ NFGA), therefore, σ |= P.