A canonical form based decision procedure and model checking approach for propositional projection temporal logic

Zhenhua Duan, Cong Tian *, Nan Zhang
ICTT and ISN Laboratory, Xi’an University, Xi’an, 710071, PR China

**A R T I C L E   I N F O**

Article history:
Received 31 October 2014
Received in revised form 27 June 2015
Accepted 24 August 2015
Available online 10 September 2015

Keywords:
Decision procedure
Propositional projection temporal logic
Büchi automata
Model checking

**A B S T R A C T**

This paper proposes a Canonical Form (CF) for chop formulas of Propositional Projection Temporal Logic (PPTL). Based on CF, an improved algorithm for constructing Labeled Normal Form Graph (LNFG) of a PPTL formula is presented. This improvement leads to a better decision procedure for PPTL with infinite models. In addition, a transformation from LNFGs to Generalized Büchi Automata (GBA) and then Büchi Automata (BA) is formalized. Thus, a SPIN based model checking approach is generalized for PPTL. To illustrate how these algorithms work, several examples are given.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

Model checking is a useful approach for verifying programs [1,2]. The usually used temporal logics for defining properties are Linear Temporal Logic (LTL) and Computing Tree Logic (CTL) [3,4]. However, as is well known, the expressive power of both LTL and CTL is limited such that some necessary properties of programs cannot be verified. In fact, the expressive power of these two logics is less than full regular expressions. Therefore, a stronger temporal logic with full regular expressive power is desired in program verification.

Propositional Projection Temporal Logic (PPTL) [7,10,5], whose expressive power is equal to full regular expressions [11], subsumes Propositional Interval Temporal Logic (PITL) [9]. It is a useful logic in the specification and verification of concurrent systems [17,18]. In the past years, a decision procedure based on Labeled Normal Form Graph (LNFG) was given for PPTL formulas [25] and is improved in [6,13]. Specifically, finiteness of LNFGs is proved in the former and a practical constructing algorithm of LNFGs is formalized in the latter. The decision algorithm is actually to find out a model in the LNFG of a formula. Given a PPTL formula \( P \), to check whether or not it is satisfiable, we first try to construct its LNFG, then look for a model in the LNFG of the formula. If a model is found out for \( P \) in its LNFG, \( P \) is satisfiable, otherwise, it is unsatisfiable. Based on the decision procedure, a model checking approach based on SPIN [15] for PPTL is proposed in [8]. Actually, to verify a property of a system with model checking, the property can be specified by a PPTL formula \( Q \), while the system is modeled with a transition system such as Kripke structure or an automaton \( M \). In the next step, we first transform \( \neg Q \) to an LNFG \( G \), then we check whether or not \( M \cap G \) is empty.

* This research is supported by National Natural Science Foundation of China under Grant Nos. 61133001, 61322202, 61420106004, 61272117, and 91448201.

* Corresponding author.

E-mail address: cttIAN@mail.xidian.edu.cn (C. Tian).

http://dx.doi.org/10.1016/j.tcs.2015.08.039
0304-3975/© 2015 Elsevier B.V. All rights reserved.
In this paper, we further improve the existing constructing algorithm of LNFGs for a PPTL formula in the following two aspects (see Section 2 for details of notations such as len($k$), fin($l_k$) etc.): (1) for a simple chop formula, $P; Q$, if $P = P' \land \text{len}(k), k \in N_0$, we do not need to add a fin label to the formula since $P$ is a terminal formula; (2) for a canonical form of chop formulas, $P = (P_1; P_2) \land Z$, where $P_1$ and $P_2$ are also canonical chop formulas, and $Z$ is not a chop formula, unlike in [6], we need only to add a fin label into the formula $P_1 \land \text{fin}(l_k); P_2$ without further concerning chop constructs inside the formulas $P_1$. In this way, less fin labels are required during the construction process of an LNFG. Thus, the constructing algorithm of LNFGs for PPTL formulas can be simplified, leading to a simpler decision procedure. In addition, an LNFG can be transformed to a Generalized Büchi Automaton (GBA) [14], and a GBA can further be transformed to a Büchi Automaton (BA) [14]. Therefore, when a system is modeled by an automaton $M$ and a property of the system is specified by a PPTL formula $P$, the LNFG $G$ of the formula $\sim P$ is constructed, then $G$ is transformed into a GBA $A$ and further transformed into a BA $A'$. To check whether or not $M \models P$ amounts to checking if $M \cap A' = \emptyset$. This can be done by utilizing model checker SPIN [15]. Thus, we can obtain a SPIN based model check approach for PPTL.

The paper is organized as follows. The next section briefly introduces PPTL, including its syntax and semantics. Section 3 discusses chop formulas. In particular, a canonical form of chop formulas is presented. In Section 4, an improved constructing algorithm of LNFGs for PPTL formulas is formalized. Some examples are given to show how the algorithm works. Section 5 focuses on SPIN based model checking approach for PPTL formulas. In particular, the transformation algorithms from an LNFG to a GBA and then to a BA are presented in details. Finally, conclusion and future research are drawn in Section 6.

2. Propositional project temporal logic

Propositional Projection Temporal Logic (PPTL) [10,7] is an extension of Propositional ITL (PITL) [9] with infinite models and a new projection construct. Let Prop be a countable set of atomic propositions and $B = \{\text{true}, \text{false}\}$ the boolean domain. Usually, we use lowercase letters like $p$, $q$ and $r$, possibly with subscripts, to denote atomic propositions and capital letters like $P$, $Q$ and $K$, possibly with subscripts, to represent general PPTL formulas. Formulas of PPTL are defined by the following grammar:

$$P ::= p \mid \neg P \mid P_1 \land P_2 \mid \bigcirc P \mid (P_1, \ldots, P_m) \text{pr}_j P \mid P^+$$

where $p \in \text{Prop}$, $\bigcirc$ (next), $+$ (chop-plus) and $\text{pr}_j$ (projection) are temporal operators. $\neg$ and $\land$ are similar as in the classical propositional logic.

We define a state $s$ over Prop to be a mapping from Prop to $B$, $s : \text{Prop} \rightarrow B$. We write $s[p]$ to denote the valuation of $p$ at state $s$. An interval $\sigma = [s_0, s_1, \ldots]$ is a non-empty sequence of states, which can be finite or infinite. The length of $\sigma$, $|\sigma|$, is the number of states in $\sigma$ minus one if $\sigma$ is finite; otherwise it is $\omega$. Let $N_0$ denote the set of non-negative integers. To have a uniform notation for both finite and infinite intervals, we will use extended integers as indices, that is $N_0 = N_0 \cup \{\omega\}$, and extend the comparison operators, $=, <, \leq$, to $N_0$ by considering $\omega = \omega$, and for all $i \in N_0, i < \omega$. Moreover, we write $\leq$ as $\leq = \{\omega, \omega\}$. To simplify definitions, we will denote $\sigma$ by $\langle s_0, \ldots, s|\sigma| \rangle$, where $s|\sigma|$ is undefined if $|\sigma|$ is infinite. With such a notation, $\sigma_i [i \leq j < |\sigma|]$ denotes the sub-interval $\langle s_i, \ldots, s_j \rangle$.

To formalize the semantics of the projection construct, we need an auxiliary operator $\downarrow$. Let $\sigma = [s_0, s_1, \ldots] >$ be an interval and $s_1, \ldots, s_h$ be integers ($h \geq 1$) such that $0 \leq s_1 \leq \ldots \leq s_h \leq |\sigma|$. The projection of $\sigma$ onto $r_1, \ldots, r_h$ is the projected interval, $\sigma \downarrow (r_1, \ldots, r_h) = \downarrow s_{1}, s_{2}, \ldots, s_{h}$, where $s_1, \ldots, s_h$ are gained from $r_1, \ldots, r_h$ by deleting all duplicates. In other words, $s_1, \ldots, s_h$ is the longest strictly increasing subsequence of $r_1, \ldots, r_h$. For instance, $\langle s_0, s_1, s_2, s_3 \rangle > (0, 2, 2, 2, 3) = [s_0, s_2, s_3] >$. The concatenation $\cdot$ of a finite interval $\sigma = [s_0, s_1, \ldots, s|\sigma| >$ with another interval $\sigma' = [s'_0, s'_1, \ldots, s'|\sigma'| >$ is represented by $\sigma \cdot \sigma' = [s_0, s_1, \ldots, s|\sigma|, s'_0, s'_1, \ldots, s'|\sigma'| >$ (not sharing any states).

An interpretation is a tuple $I = (\sigma, k, j)$, where $\sigma = [s_0, s_1, \ldots]$ is an interval, $k$ is a non-negative integer, and $j$ is an integer or $\omega$ such that $0 \leq k \leq j \leq |\sigma|$. We write $I(s, k, j)$ to mean that a formula is interpreted over a subinterval $\sigma(k,j)$ with the current state being $s_k$. We utilize $I^\text{prop}_k$ to stand for the state interpretation at state $s_k$. The satisfaction relation $I \models$ for formulas is given as follows:

$$I \models p \quad \text{iff} \quad s_0[p] = I^\text{prop}_k[p] = \text{true}$$

$$I \models \neg P \quad \text{iff} \quad I \not\models P$$

$$I \models P_1 \land P_2 \quad \text{iff} \quad I \models P_1 \text{ and } I \models P_2$$

$$I \models \bigcirc P \quad \text{iff} \quad k < j \text{ and } (\sigma, k + 1, j) \models P$$

$$I \models (P_1, \ldots, P_m) \text{pr}_j P \quad \text{iff there exist integers } r_0, \ldots, r_m, \text{ and } k = r_0 \leq \ldots \leq r_m \leq \text{fin } k \leq m \text{ such that }$$

$$(1) \quad r_m < j \text{ and } \sigma^j = \sigma \downarrow (r_0, \ldots, r_m) \cup (r_{m+1}, \ldots, j)$$

$$(2) \quad r_m = j \text{ and } \sigma^j = \sigma \downarrow (r_0, \ldots, r_m) \text{ for some } 0 \leq k \leq m$$

$$I \models P^+ \quad \text{iff there are finitely many integers } r_0, \ldots, r_n \text{ and } k = r_0 \leq r_1 \leq \ldots \leq r_{n-1} \leq r_n \text{ such that }$$

$$(\sigma, r_{n+1}, r_1) \models P \text{ for all } 1 \leq l \leq n \text{ or } j = \omega \text{ and there are infinitely many integers }$$

$$k = r_0 \leq r_1 \leq r_2 \leq \ldots \text{ such that } \lim_{i \rightarrow \infty} r_1 = \omega \text{ and } (\sigma, r_{n+1}, r_1) \models P \text{ for all } l \geq 1.$$
For convenience, some derived formulas from elementary PPTL formulas are shown below, which are explained in [7,10]. The abbreviations true, false, $\lor$, $\rightarrow$ and $\leftarrow$ are defined as usual.

\[
\begin{align*}
\varepsilon & \overset{\text{def}}{=} \neg \bigcirc \text{true} & \text{more} & \overset{\text{def}}{=} \neg \varepsilon \\
\nabla P & \overset{\text{def}}{=} (\text{true}, P) \lor P & \nabla \neg P & \overset{\text{def}}{=} \neg \bigcirc \neg P \\
\text{fin}(P) & \overset{\text{def}}{=} (\varepsilon \rightarrow P) \land \text{halt}(P) & \overset{\text{def}}{=} \varepsilon \rightarrow \neg \varepsilon \\
\text{keep}(P) & \overset{\text{def}}{=} (\varepsilon \rightarrow P) \lor \text{rem}(P) & \nabla \neg \text{rem}(P) & \overset{\text{def}}{=} \varepsilon \rightarrow \neg \varepsilon \\
\text{fin} & \overset{\text{def}}{=} \nabla \varepsilon & \text{inf} & \overset{\text{def}}{=} \nabla \text{more} \\
\text{fin}^* & \overset{\text{def}}{=} P \lor \varepsilon & \text{len}(n) & \overset{\text{def}}{=} \begin{cases} 
\varepsilon & \text{if } n = 0 \\
= n - 1 & \text{if } n > 1
\end{cases}
\end{align*}
\]

Usually, $\vDash (P \leftrightarrow Q)$ is represented by $P \equiv Q$ (strong equivalence), meaning that $P$ and $Q$ have the same truth value at all states of any models while $\vDash (P \rightarrow Q)$ is denoted by $P \supset Q$ (strong implication), stating that $P \rightarrow Q$ is true at all states of any models. The following are some useful logic laws. Here $w$ is a state formula. The proofs of the logic laws can be found in [7].

\[
\begin{align*}
L_1 & : (P \land Q) \equiv \neg \bigcirc \neg P \land \neg \bigcirc \neg Q \\
L_2 & : (P \lor Q) \equiv \neg \bigcirc \neg P \lor \neg \bigcirc \neg Q \\
L_3 & : (P \lor Q) \equiv \neg \bigcirc \neg P \lor \neg \bigcirc \neg Q \\
L_4 & : (P \land Q) \equiv \neg \bigcirc \neg P \land \neg \bigcirc \neg Q \\
L_5 & : (P \lor Q) \equiv (R \lor P) \lor (R \lor Q) \\
L_6 & : (P \lor Q) : R \equiv (P \lor Q) \lor (R \lor Q) \\
L_7 & : (P \lor Q) \equiv P \lor Q \\
L_8 & : (\neg P) \equiv \neg P \lor \neg P \\
L_9 & : \text{more} \land \neg \bigcirc \neg P \equiv \text{more} \land \neg \bigcirc \neg P \\
L_10 & : (\neg \bigcirc \neg P) \equiv \neg \bigcirc \neg P \\
L_11 & : (P \lor Q) \equiv (P \lor Q) \\
L_12 & : (w \land (P \lor Q)) \equiv (w \land P) \lor Q \\
L_13 & : (Q \land P) \equiv Q \\
L_14 & : (\varepsilon \land P) \equiv Q \\
L_15 & : (P_1, \ldots, P_m) \land (P \lor Q) \equiv (P_1, \ldots, P_m) \lor (P \lor Q) \\
L_16 & : (P, \varepsilon) \land (P \lor Q) \equiv (P \land \varepsilon) \land Q \\
L_17 & : (P_1, \ldots, P_m) \land (P \lor Q) \equiv (P_1, \ldots, P_m) \lor (P \lor Q) \\
L_18 & : (P_1, \ldots, P_m) \lor (P_1, \ldots, P_m) \lor (P \land \varepsilon) \lor (P \land \varepsilon) \\
L_19 & : (P_1, \ldots, P_m) \lor (P \land \varepsilon) \lor (P \land \varepsilon) \lor (P \land \varepsilon) \\
L_20 & : (P_1, \ldots, P_m) \lor (P \land \varepsilon) \lor (P \land \varepsilon) \lor (P \land \varepsilon) \\
L_21 & : (P_1, \ldots, P_m) \lor (P \land \varepsilon) \lor (P \land \varepsilon) \lor (P \land \varepsilon) \\
L_22 & : (P_1, \ldots, P_m) \lor (P \land \varepsilon) \lor (P \land \varepsilon) \lor (P \land \varepsilon) \\
L_23 & : (P_1, \ldots, P_m) \lor (P \land \varepsilon) \lor (P \land \varepsilon) \lor (P \land \varepsilon) \\
\end{align*}
\]

By the derived formulas and logic laws, we can further prove the following conclusions [25,7]:

\[
\begin{align*}
\text{fin}(P) & \equiv P \land \varepsilon \land \neg \bigcirc \text{fin}(P) \land \text{keep}(P) & \equiv \varepsilon \land P \land \neg \bigcirc \text{keep}(P) \\
\text{halt}(P) & \equiv P \land \varepsilon \land \neg \bigcirc \text{halt}(P) \land \text{rem}(P) & \equiv \varepsilon \land \neg \bigcirc \text{rem}(P) \\
\text{inf} & \equiv \text{true} & \text{false} & \equiv \varepsilon \lor \bigcirc \text{true}
\end{align*}
\]

3. Chop formulas

Chop formulas play important roles in our constructing algorithm of LNFGs for PPTL formulas. Although chop formulas are discussed in [6], a formal analysis has not been given. In this section, we present the definition of chop formulas and their canonical form (CF) so that some important conclusions are achieved and the constructing algorithm of LNFGs can be simplified.

A chop formula $R_C$ is inductively defined as follows [6]:

\[
R_C ::= P; Q \mid R_C \land P
\]

where $P$ and $Q$ are any PPTL formulas. Further, $P; Q$ is called a chop component of chop formula $R_C$. For instance, $(p; q) \land \neg r$ and $(\neg q; r)$ are chop formulas while $\neg (p; q)$ and $(p; q)$ are not, where $p, q$, and $r$ are atomic propositions. Note that $P; Q$ is a chop formula but $\neg (P; Q)$ is not.
Here, we give an equivalent definition of chop formulas so that an inductive proof can easily proceed.

\[ R_c ::= P_1; P_2 | R_c \land R_c | R_c \land P | R_c; R_c | P; R_c \]

(2)

where \( P, P_1 \) and \( P_2 \) are PPTL formulas without chop being the main operator.

The above two definitions are obviously equivalent since each rule of one definition can be expressed by the other. Based on (2), we can further define a canonical form of chop formulas. The canonical form of chop formulas is useful in constructing LNFGs of PPTL formulas so that the satisfiability of PPTL formulas can be checked. In what follows, we use \( P, Q, \) or \( X \) to denote an arbitrary PPTL formula.

**Definition 1.** For a chop formula \( P_1; \ldots; P_m \), where \( 2 \leq m \leq N \), and each \( P_i, 1 \leq i \leq m, \) is a formula without chop being the main operator, its right most chop component (RC) denoted as \( R_{rc} \) is defined by:

\[ R_{rc} \equiv (P_1; \ldots; P_{m-1}); P_m \]

(3)

The conjunction of a right most chop component \( R_{rc} \) and an arbitrary PPTL formula \( X \) is called the right most chop formula (RCF), denoted by \( R_{rcf} \). That is,

\[ R_{rcf} \equiv R_{rc} \land X \]

(4)

Thus, a right most chop component is also a right most chop formula, but not vice versa. For instance, formula \( (p; \neg q); \Box r \), where \( p, q \) and \( r \) are atomic propositions, is not only a right most chop component but also a right most chop formula. Nevertheless, \( ((p; \neg q); \Box r) \land \Box p \) is only a right most chop formula but not a right most chop component. Further, formulas \( \neg(p; \neg q); \Box r \) and \( \neg(p; \neg q; \Box r) \) are neither a right most chop component nor a right most chop formula. In addition, formula \( p; q \) is a right most chop component and \( (p; q) \land X \) (\( X \) is not true) a right most chop formula.

As a matter of fact, any chop formula can be equivalently transformed to a RC or RCF form.

**Theorem 1.** Any chop formula \( R_c \) can be transformed to a right most chop formula.

**Proof.** We just need to prove that any chop formula \( R_c \) generated by (2) can be equivalently transformed as a right most chop formula. The proof proceeds by induction on the structure of chop formulas.

**Base:** If \( R_c = P_1; P_2 \), it is already a right most chop formula since \( P_1 \) and \( P_2 \) are not chop formulas.

**Induction:** Suppose that \( R_{c1} \) and \( R_{c2} \) are chop formulas in RCF. That is \( R_{c1} \) and \( R_{c2} \) have been equivalently transformed as follows:

\[ R_{c1} \equiv ((P_1; \ldots; P_{m-1}); P_m) \land X_1 \]

\[ R_{c2} \equiv ((Q_1; \ldots; Q_{n-1}); Q_n) \land X_2 \]

- In case \( R_c \equiv R_{c1} \land R_{c2} \), it has:
  
  \[ R_c \equiv R_{c1} \land R_{c2} \equiv ((P_1; \ldots; P_{m-1}); P_m) \land X_1; ((Q_1; \ldots; Q_{n-1}); Q_n) \land X_2 \]

- In case \( R_c \equiv R_{c1} \land P \), where \( P \) is not a chop formula, we have:
  
  \[ R_c \equiv R_{c1} \land P \equiv ((P_1; \ldots; P_{m-1}); P_m) \land X_1; P \]

- In case \( R_c \equiv P \land R_{c1} \)
  
  \[ R_c \equiv P \land R_{c1} \equiv P \land ((P_1; \ldots; P_{m-1}); P_m) \land X_1 \]

- In case \( R_c \equiv R_{c1} \land P \) or \( R_c \equiv R_{c1} \land R_{c2} \), the conclusion holds obviously by the definition of right most chop formulas.

Therefore, any chop formula can be equivalently transformed as a right most chop formula. \( \square \)

The above theorem only requires that a chop formula \( R_c \) is written in an RC or RCF in the form of \( R_{c1}; R_{c2} \) or \( (R_{c1}; R_{c2}) \land X \) ignoring sub-formulas inside \( R_{c1} \) and \( R_{c2} \). If we further require that each \( R_{ci} (1 \leq i \leq 2) \) is recursively written as a RC or RCF unless it is a non-chop formula, we can obtain a canonical form (CF) for a chop formula.

**Definition 2.** Canonical form \( R_{cf} \) of a chop formula is defined by:

\[ R_{cf} ::= (P_1; \ldots; P_{m-1}); P_m \land X \mid R_{cf} \land R_{cf} \mid (R_{cf}; \ldots; R_{cf}) \land X \]

where \( P_1, \ldots, P_m, \) and \( X \) are arbitrary non-chop formulas. \( \square \)
Actually, any chop formula can be re-written in its canonical form since we can recursively transform its sub-formulas inside using the underlying constructing algorithm given in Theorem 1. Therefore, we have the following corollary.

**Corollary 2.** Any chop formulas can be transformed to its canonical form.

The canonical form of a chop formula allows us to save fin labels when constructing an LNFG of a PPTL formula.

**Theorem 3.** Let \( Q \) be a chop formula. Its canonical form

\[
Q = \bigwedge_{i=1}^{n} (\text{fin}(l^i); R^i_{\text{cf},m}) \land X
\]

is satisfiable iff

\[
Q' = \bigwedge_{i=1}^{n} ((\text{fin}(l^i); R^i_{\text{cf},m}) \land \text{fin}(l^i)); R^i_{\text{cf},m}) \land X
\]

is satisfiable.

**Proof.** We only need to prove that for any \( h, 1 \leq h \leq n, R = (R^h_{\text{cf},1}; \ldots; R^h_{\text{cf},(m-1)}); R^h_{\text{cf},m} \) is satisfiable iff \( R' = (R^h_{\text{cf},1}; \ldots; R^h_{\text{cf},(m-1)}; \text{fin}(l^h)); R^h_{\text{cf},m} \) is satisfiable. For convenience, we use \( R_c \) to denote \( R_{\text{cf},1}; \ldots; R_{\text{cf},m} \).

**Case 1:** \( R_c \) is a terminable formula.

In this case, there exists a finite reduced sequence: \( R_c, R^1_c, R^2_c, R^3_c, \ldots, R^k_c \) such that \( R_c \) is eventually reduced to \( R^k_c \) and \( \epsilon \). This leads to a finite reduced sequence for \( R, i.e., R, R^1, \ldots, R^k \), such that \( R^k \equiv (R^k_c \land \epsilon); R^h_{\text{cf},m} \equiv R^k \land R^h_{\text{cf},m} \). Similarly, we have \( R^k \equiv (R^k_c \land \epsilon \land \text{fin}(l^h)); R^h_{\text{cf},m} \equiv R^k \land R^h_{\text{cf},m} \). We define \( P^{\text{fin}} \equiv Q \) if \( P \equiv Q \) without considering \( \text{fin}(l) \) and \( l_k \) labels in \( P \) and \( Q \). Therefore, \( R^k_{\text{cf},m} \equiv R^k_c \).

**Case 2:** \( R_c \) is not a terminable formula.

In this case, \( R_c \) is reduced to an infinite sequence leading to a circle:

\[
R_c = R^0_c, R^1_c, R^2_c, \ldots, R^j_c, \ldots, R^k_c
\]

Here \( 0 \leq j \leq k \). On one hand, according to the semantics of chop operation, \( R \) is obviously false. Hence, \( R \) is unsatisfiable. On the other hand, \( R' \) is also reduced to an infinite path, \( R_c \land \text{fin}(l), R^1_c \land \text{fin}(l), R^2_c \land \text{fin}(l), \ldots, R^j_c \land \text{fin}(l), \ldots, R^k_c \land \text{fin}(l) \ldots, R^j_c \land \text{fin}(l) \). Thus, in the LNFG of \( R' \), for any path, there exists a circle in which each node contains a label \( \text{fin}(l) \). According to the decision procedure, \( R' \) is unsatisfiable. \( \square \)

Theorems 1 and 3 illustrate how to deal with chop formulas for constructing LNFGs. By Theorem 3, we only need to add \( \text{fin} \) labels to chop components \( R^1_c; R^2_c \) without considering the chop components inside \( R^1_c \).

Generally, we can move negation operators to the front of atomic propositions, chop, or projection formulas by means of distributive laws on conjunction and disjunction operations. Moreover, using the distributive laws of disjunction operator on chop and projection operators, any PPTL formula can further be written as the form \( \bigvee_i P_i \), where \( P_i \) is either a chop formula in its canonical form or a non-chop formula. These auxiliary work can be done in the procedure PPRE [6]. This is the reason why we take each disjunctive component into account and deal with each chop formula with its canonical form (conjunction form) while constructing the LNFG for a PPTL formula.

4. Decision procedure

In this section, an improved constructing algorithm of LNFGs for interval based temporal logics is formalized. Further, three examples are given to show the difference between LNFGs constructed by the existing algorithm in [6] and the improved one, respectively.

4.1. The existing algorithm for constructing LNFGs

The algorithm in [6] is a novel approach by rewriting a chop component \( P; Q \) as \( P \land \text{fin}(l_k) \); \( Q \) for some \( k \in \mathbb{N} \) whenever a new chop formula is encountered. Further, \( P \land \text{fin}(l_k) \) if \( P \equiv Q \) without considering \( \text{fin}(l) \) and \( l_k \) labels in \( P \) and \( Q \). A node added with the second \( \text{fin}(l_k) \) is placed in \( \text{CL}'(P) \) and a node with the third \( \text{fin}(l_k) \) is forced to point to the first one in \( \text{CL}(P) \). This prevents from adding \( \text{fin} \) labels to the same node more than twice.

In the following, three examples are given to show LNFGs constructed by the algorithm in [6].
Example 1. LNFG of formula $P \equiv \square(p_1 \land \square^2:q) \land \square \circ p_2 \lor \square(p_1 \land \text{more};q) \land \square \circ p_3$.

$P$ is in the form of $P_1 \lor P_2$, where $P_1 \equiv \square(p_1 \land \square^2:q) \land \square \circ p_2$ and $P_2 \equiv \square(p_1 \land \text{more};q) \land \square \circ p_3$. By the algorithm in [6], as shown in Fig. 1, there are two root nodes in the LNFG $G = (CL(P), EL(P), V_0, L = [L_1, \ldots, L_m])$ of formula $P$. Here, $V_0 = \{n_0, n_1\}$, and

$$CL(P) = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}$$

$$EL(P) = \{(n_0, p_1, n_2), (n_1, p_1, n_3), (n_2, p_1 \land p_2, n_4), (n_3, p_1 \land p_3, n_5), (n_3, p_1 \land p_3 \land q_1 \land l_2, n_3), (n_4, p_1 \land p_2 \land q_1 \land l_1, n_6), (n_5, p_1 \land p_3, n_3), (n_6, p_1 \land p_2 \land q_1 \land l_3, n_4)\}.$$

$L = \{L_1 = \{n_2\}, L_2 = \{n_3, n_5\}, L_3 = \{n_4\}, L_4 = \{n_5\}, L_5 = \{n_6\}\}$

Example 2. LNFG of formula $Q \equiv (p_1 \land \text{more}; q_1 \land \epsilon) \land \text{more}; p_2 \land \text{more}; q_2; \square(\text{more} \land (p; q))$.

Using the existing constructing algorithm, the LNFG $G = (CL(Q), EL(Q), V_0, L = [L_1, \ldots, L_m])$ of formula $Q$ is constructed as depicted in Fig. 2, where

$$V_0 = \{n_0\}$$

$$CL(Q) = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}$$

$$EL(Q) = \{(n_0, p_1, n_1), (n_1, \text{true}, n_1), (n_1, q_1 \land p_2 \land l_1, n_2), (n_2, \text{true}, n_2), (n_2, q_2 \land l_2, n_3), (n_2, q_2 \land p \land l_2, n_4), (n_3, \text{true}, n_3), (n_3, p \land l_3, n_4), (n_4, q \land p \land l_4, n_5), (n_5, p, n_4), (n_5, q \land p, n_5), (n_6, p, n_4), (n_6, q \land p \land l_5, n_5)\}.$$

$L = \{L_1 = \{n_0, n_1\}, L_2 = \{n_2\}, L_3 = \{n_3\}, L_4 = \{n_4, n_5\}, L_5 = \{n_6\}\}$

Example 3. LNFG of formula $R \equiv \langle \text{more}; p \rangle^* \land \square(q \land \square^2 \langle \text{more}; p \rangle)$.

Constructed by the existing algorithm, the LNFG $G = (CL(R), EL(R), V_0, L = [L_1, \ldots, L_m])$ of formula $R$ is depicted in Fig. 3, where

$$V_0 = \{n_0\}$$

$$CL(R) = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$$

$$EL(R) = \{(n_0, n_1), (n_1, q \land p \land l_1, n_2), (n_1, q, n_3), (n_2, q \land l_3, n_4), (n_2, q, n_5), (n_3, q \land p \land l_1, n_6), (n_3, q, n_7), (n_4, q \land p \land l_5, n_5), (n_4, q, n_4), (n_5, q \land l_3, n_4), (n_5, q, n_5), (n_6, q \land l_9, n_4), (n_6, q, n_5), (n_7, q \land p \land l_1, n_5), (n_7, q, n_4)\}.$$

$L = \{L_1 = \{n_1, n_3, n_7\}, L_2 = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}, L_3 = \{n_2, n_5\}\}

L_4 = \{n_2, n_4, n_5\}, L_5 = \{n_3, n_6, n_7\}, L_6 = \{n_4\}, L_7 = \{n_4\}, L_8 = \{n_5\}, L_9 = \{n_6\}, L_{10} = \{n_6\}, L_{11} = \{n_7\}\}$
4.2. The improved algorithm for constructing LNFGs

The improved algorithm for constructing LNFG of a PPTL formula is shown in Algorithm Lnfg. Basically, we improve the existing algorithm in two aspects: (1) for a chop formula, $P; Q$, if $P \equiv P' \land \text{len}(k), k \in \mathbb{N}_0$, we do not need to add a fin label to the formula since $P$ is a terminal formula; (2) for a canonical form of chop formulas, $P \equiv (P_1; P_2) \land Z$, where $P_1$ and $P_2$ are also in the canonical forms, and $Z$ is not a chop formula, unlike in [6], we need only add a fin label into the formula $P_1 \lor \text{fin}(k); P_2$ without further concerning chop constructs inside formula $P_1$. In this way, less fin labels are required when constructing LNFGs. We use the same examples given in Section 4.1 to illustrate difference between the two algorithms, intuitively.

The LNFG of formula $P \equiv [\square(p_1 \land \square^2 q; q)] \lor \square(p_1 \land \text{more}; q) \lor \square \circ p_3$ constructed by the improved algorithm is illustrated in Fig. 4, where $V_0 = \{n_0, n_1\}$, and

$CL(P) = [n_0, n_1, n_2, n_3, n_4, n_5]$

$EL(P) = [(n_0, n_1, n_2), (n_1, p_1, n_3), (n_2, p_1 \land p_2, n_4), (n_3, p_1 \land p_3, n_5),$

$(n_3, p_1 \land p_3 \land q_1 \land l_1, n_3), (n_4, p_1 \land p_2 \land q_1, n_4), (n_5, p_1 \land p_3, n_3)]$

$L = \{L_1 = \{n_3, n_5\}, L_2 = \{n_5\}\}$

Compared with the LNFG presented in Fig. 1, two labels as well as one node and one edge are saved.
Algorithm LnsF: Constructing LNFG of a PPTL formula.

/* Input: P = ∨/i Pi (0 ≤ i < n) is a PPTL formula */
/* Output: LNFG of P, G = ⟨CL(P), EL(P), V0, L⟩ = {L1, ..., Lm} */

Begin function

1: V0 = CL(P) = {Pi | Pi appears in ∨/i Pi}; Mark[Pi] = 0 for each i;
2: IF(P) = ∅; AddD = AddN = 0; k = 1; L = ∅; CL(P) = ∅; /* Initialization */
3: while ∃e ∈ (CL(P) ∪ EL(P))\{e\} and Mark[k] = 0
5: if R ≠ ∅ then /* R is false if R ∧ ¬ appears in it */
6: if R = (∨/i Rj) ∨ Z with Rj ∈ Pi; Qj in canonical form and Z is not a chop formula then
7: for i = 1 to n /* adding labels to chop formulas */
8: if P = P' ∧ len(n) for n ∈ N then continue;
9: else if Rj has been rewritten with a [fin(i)] (1 ≤ j < k)
10: then Lj = Lj ∪ {R} /* incorporating R to corresponding Lj */
11: else Rewrite Rj as Pj ∧ [fin(i)]; Qj: Lq = Lj ∪ {R}; k = k + 1;
12: end for
13: end if
14: Qj = NF(R); Mark[k] = 1; /* rewriting R into its normal form */
15: case /* deciding whether future products or terminal products are contained in NF */
16: Q = B ∨/m Qn ∧ e; AddD = 1;
17: Q = B ∨/m Qn ∧ e; AddD = 1;
18: Q = B ∨/m Qn ∧ e; AddD = 1;
19: end case
20: if AddD == 1 then
21: for i = 1 to m /* dealing with each Qn */
22: if Qj ≠ false then
23: if P ∈ CL(P) then CL(P) = CL(P) ∪ {e}; end if
24: EL(P) = EL(P) ∪ (R, Qj, e);
25: end if
26: end for
27: AddD = 0;
28: end if
29: if AddN == 1 then
30: for j = 1 to n /* dealing with each Qj */
31: if Qj ≠ false then
32: if Qj ∈ CL(P) then
33: if ∃P ∈ CL(P) such that Qj = P
34: if P ∈ CL(P) such that Qj = P
35: else CL(P) = CL(P) \{Qj\}; EL(P) = EL(P) ∪ (R, Qj, Qj)
36: else CL(P) = CL(P) \{Qj\}; EL(P) = EL(P) \{Qj\}; mark(Qj) = 0;
37: else CL(P) = CL(P) \{Qj\}; EL(P) = EL(P) \{Qj\}; mark(Qj) = 0;
38: end if
39: end if
40: end if
41: end for
42: AddN = 0;
43: end if
44: end if
45: end while
46: L = B ∨/m Lj; CL(P) = CL(P) ∪ CL';
47: while ∃R ∈ CL(P), such that R is not e and has no edges departing from
48: CL(P) = CL(P) \{R\}; EL(P) = EL(P) \{R, R, R\}; L = L \{L1, ... , Lm \{R\}];
49: end while
50: return G = (CL(P), EL(P), V0, L);

End function

Fig. 4. LNFG of P.
Next, to construct the LNFG of formula \( Q = (p_1 \land \text{more} \land q_1 \land e) \land \text{more} \land p_2 \land \text{more} \land q_2; \Box(\text{more} \land (p; q)) \), we first rewrite \( Q \) to its canonical form since it is a chop formula. Then we construct its LNFG \( G = (\text{CL}(Q), \text{EL}(Q), V_0, \mathbb{L} = \{L_1, \ldots, L_m\}) \) as shown in Fig. 5 where

\[
V_0 = \{n_0\}
\]

\[
\text{CL}(Q) = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}
\]

\[
\text{EL}(Q) = \{(n_0, p_1, n_1), (n_1, \text{true}, n_1), (n_1, q_1 \land p_2, n_2), (n_2, \text{true}, n_2), (n_2, q_2, n_3), (n_2, q_2 \land p \land l_1, n_4), \}
\]

\[
(n_2, q_2 \land q \land p \land l_1, n_5), (n_3, \text{true}, n_3), (n_3, p \land l_1, n_4), (n_3, q \land p \land l_1, n_5), (n_4, p, n_6), (n_4, q \land p \land l_2, n_5),
\]

\[
(n_5, p, n_4), (n_5, q \land p, n_5), (n_6, p, n_4), (n_6, q \land p \land l_3 \land l_5), \}
\]

\[
\mathbb{L} = \{L_1 = \{n_0, n_1, n_2, n_3\}, L_2 = \{n_4, n_5\}, L_3 = \{n_6\}\}
\]

Compared with the LNFG shown in Fig. 2, two fin labels are saved.

Finally, for formula \( R = (\text{more} \land p)^\ast \land \Box(q \land \Box q^2 \land \text{more} \land p) \), since \( R \) contains neither the form of \( P' \land \text{len}(n) \) for \( n \in N_0 \), nor its canonical form, the new LNFG is the same as one constructed in Fig. 3.

4.3. Improved decision procedure

The LNFG of a formula \( P \) can be constructed by Algorithm \text{LnfG}. Further, \( P \) is satisfiable if and only if there exist finite or infinite models in the LNFG of \( P \). Consequently, a decision procedure for checking the satisfiability of a PPTL formula \( P \) can be formalized based on the LNFG of \( P \). Algorithm \text{Check} shows the algorithm for checking whether or not a PPTL formula \( P \) is satisfiable where an auxiliary function \text{Decision} is invoked to decide whether or not an LNFG \( G \) has models. Specifically, if there is \( e \) node in the LNFG \( G \), \( G \) has finite models; and if there is an infinite path satisfying certain conditions in the LNFG \( G \), \( G \) has infinite models. Here we use a global variable \( M \) to store the models of \( G \).

According to the LNFGs of the three formulas given in Section 4.2, we can check their satisfiability. First, \( P \) is satisfiable since there is a strongly connected component \( [n_4] \) where \( n_4 \) is not annotated by any fin labels as shown in Fig. 4. Second, \( Q \) is satisfiable since a strongly connected component \( [n_4, n_5, n_6] \) where no common labels are annotated at \( n_4, n_5, \) and \( n_6 \) as depicted in Fig. 5 can be found out. Finally, \( R \) is unsatisfiable since there is no \( e \) node in the LNFG of \( R \), and there is only one strongly connected component with two nodes \( n_4 \) and \( n_5 \) but \( n_4 \) and \( n_5 \) have common labels \( \text{fin}(l_2) \) and \( \text{fin}(l_4) \), as shown in Fig. 3.

4.4. A dynamic algorithm

As discussed above, when we check the satisfiability of a PPTL formula \( P \), we first construct a complete LNFG \( G \) of \( P \) by Algorithm \text{LnfG}. Then we check whether or not \( G \) has a finite or infinite model. Alternatively, we may combine the constructing and checking processes together. That is, during the construction of the LNFG of \( P \), whenever we find a finite or infinite model of \( P \), we can terminate the processes and decide that \( P \) is satisfiable without continuing constructing the whole LNFG of \( P \). Keep this in mind, in the following, we give a dynamic algorithm, Algorithm \text{Check-along-construct}, for checking the satisfiability of PPTL formulas.
Algorithm 3 Decision Deciding whether $G$ has models.

Function Decision($G$)
/* precondition: $G = (CL(P), E(P), V_0, L = \{L_1, \ldots, L_m\})$ is an LNFG of PPTL formula $P$ */
/* postcondition: Decision($G$) returns 0, 1 or 2 meaning that $G$ has no any models, has finite or infinite models respectively */

begin function
1: $M = \Phi$; /* initialization */
2: if there exists an $s$ node in $CL(P)$ then
3: $M = M \cup \{P_s\}$; /* $P_s$ denotes a finite path of $G$ */
4: return 1;
5: end if
6: $S = Tarjan(G);/ s = S$ is the set of strongly connected components of $G$. */
7: for each $s$ in $S$ do
8: if $s$ has only one node and the node has a self-loop and the node is not labeled by $fin$ then
9: $M = M \cup \{s\};$
10: return 2;
11: else if $s$ has more than one node and these nodes have no common labels then
12: $M = M \cup \{s\};$
13: return 2;
14: end if
15: end for
16: return 0;
end function

Algorithm 4 Check Checking the satisfiability of $P$.

Function Check($P$)
/* precondition: $P$ is a PPTL formula */
/* postcondition: Check($P$) checks whether or not formula $P$ is satisfiable */

begin function
1: $G = Lsre(P);$ 
2: mode = Decision($G$); 
3: if mode == 1 then 
4: return $P$ is satisfiable with finite models; 
5: else if mode == 2 then 
6: return $P$ is satisfiable with infinite models; 
7: else return $P$ is not satisfiable; 
8: end if 
end function

We compare Algorithm Check-along-construct with Algorithm Check by the following four formulas:

$$\theta_1 = (p_1 \land \Box^2 \epsilon; q) \land \Box \Box p_2 \lor \Box(p_1 \land \text{more}; q) \land \Box \Box p_3$$

$$\theta_2 = (p_1 \land \text{more}; q_1 \land \epsilon) \lor \text{more}; p_2 \land \text{more}; q_2; \Box(\text{more} \land (p; q))$$

$$\theta_3 = (\text{more}; p)^* \land \Box(q \land \Box^2 \text{more}; p)$$

$$\theta_4 = \neg((3S; \neg(\neg(\Box r; \Box q); \Box p))$$

The comparison results are depicted in Table 1. As we can see, (1) when the model of the LNFG of a PPTL formula appears in an early time as $\theta_4$ shows, the algorithm Check-along-construct performs better. (2) When the LNFG of a PPTL formula has no model as $\theta_3$ shows, Algorithm Check acts better. The reason is simple since Check-along-construct has to not only construct a complete LNFG of a PPTL formula but also check whether or not there is a model up to now during the constructing process.

5. Model checking approach for PPTL based on SPIN

With our model checking algorithm for PPTL, a system is modeled as a Büchi automaton (BA) $A_S$, and the property to be verified is specified by a PPTL formula $P$. To check if the system satisfying $P$ is valid, we first transform $\neg P$ to an LNFG, then the LNFG to a Generalized Büchi Automaton (GBA), and finally the GBA to a BA [12] $A_W$. Then we calculate the product automaton of $A_S$ and $A_W$. If the product automaton is empty then the system satisfying the property is valid otherwise a counterexample is given.

5.1. Transformation from an LNFG to a GBA

After we have obtained an LNFG for a PPTL formula $P$, we cannot transform the LNFG into a Büchi automaton directly because of the difference between LNFGs and Büchi automata in accepting conditions. We need transform an LNFG to a GBA first, and then to a BA eventually.
Algorithm CHECK-ALONG-CONSTRUCT Checking the satisfiability of PPTL formula P while constructing LNFC.

Function CHECK-ALONG-CONSTRUCT(P)
/* precondition: P = ϕ(P_i) (0 ≤ i) is a PPTL formula */
/* postcondition: CHECK-ALONG-CONSTRUCT(P) checks whether or not P is satisfiable */
begin function
1: V₀ = CL(P) = {P_i | P_i appears in ϕ(P_i); Mark[P_i] = 0 for each i;
2: EL(P) = ϕ; AddDE = AddDN = 0; k = 1; L₁ = ϕ; CL'(P) = P; /* initialization */
3: while ∃R ∈ (CL(P) ∪ CL'(P)) \ {ϕ} and mark[R] == 0 
4: R = PRE(R);
5: if R ≠ false then 
6: if R = (Λⁿ_i)ₖ = 0 then
7: for i = 1 to n /* adding labels to chop formulas */
8: if P = P' ∧ len(n) for n ∈ N₀ then continue;
9: else
10: then L₁ = L₁ ∪ {R} /* incorporating R to corresponding L₁ */
11: else Rewrite Rₖ as Pᵢ ∧ fin(k); Qᵢ; Lₖ = Lᵢ ∪ {R}; k = k + 1;
12: end for
end if
13: if Q = N(P) then
14: Q = NF(P); mark[R] = 1; /* rewriting R into its normal form */
15: case /* deciding whether future products or terminal products are contained in NF */
16: Q = ∨∄mᵢ₁ Qᵢ ∧ ϕ; AddDE = 1;
17: Q = ∨j=1 Qᵢ ∨ Q_j; AddN = 1;
18: Q = ∨mᵢ₁  Qᵢ ∨ Q_j; AddDE = AddN = 1;
19: end case
20: if AddDE == 1 then
21: for i = 1 to m /* dealing with each Qᵢ */
22: if Qᵢ ≠ false then
23: if ϕ ∉ CL(P) then CL(P) = CL(P) ∪ {ϕ}; end if
24: EL(P) = EL(P) ∪ (R, Qᵢ, ϕ);
25: end if
26: end for
27: AddDe = 0;
28: end if
29: if AddN == 1 then
30: for j = 1 to h /* dealing with each Qⱼ */
31: El(Qⱼ) ≠ false then
32: if Qⱼ ≠ CL(P) then
33: if ∃Pⱼ ∈ CL(P) such that Qⱼ = Pⱼ then EL(P) = EL(P) ∪ {(R, Qⱼ, Pⱼ)}
34: /* a node added with third fin(k) is forced to point to the first one in CL(P) */
35: else CL(P) = CL(P) ∪ {Qⱼ}; EL(P) = EL(P) ∪ {(R, Qⱼ, Qⱼ)}; mark[Qⱼ] = 0; 
36: /* a node added with second fin(k) is placed in CL(P) */
37: end if
38: if EL(P) = EL(P) ∪ {(R, Qⱼ, Qⱼ)} then
39: end if
40: end if
41: end for
42: AddN = 0;
43: end if
44: end if
45: mode = Decision(G = (CL(P) ∪ CL'(P), EL(P), V₀, L₁ = ∪₀≤i≤k (Lᵢ)));
46: if mode == 1 then return P is satisfiable with finite models;
47: else if mode == 2 then return P is satisfiable with infinite models;
48: end if
49: end while
50: mode = Decision(G = (CL(P) ∪ CL'(P), EL(P), V₀, L₁ = ∪₀≤i≤k (Lᵢ)));
51: if mode == 1 then return P is satisfiable with finite models;
52: else if mode == 2 then return P is satisfiable with infinite models;
53: else return P is not satisfiable;
54: end if
end function

Table 1
Comparison results.

<table>
<thead>
<tr>
<th>Algorithm Check</th>
<th>Algorithm CHECK-ALONG-CONSTRUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (s) node</td>
<td>time (s) node</td>
</tr>
<tr>
<td>graph edge</td>
<td>graph edge</td>
</tr>
<tr>
<td>θ₁ 0.031 6 7</td>
<td>θ₁ 0.031 6 6</td>
</tr>
<tr>
<td>θ₂ 0.047 7 16</td>
<td>θ₂ 0.032 7 14</td>
</tr>
<tr>
<td>θ₃ 0.078 8 15</td>
<td>θ₃ 0.141 8 15</td>
</tr>
<tr>
<td>θ₄ 151.96 162 2920</td>
<td>θ₄ 0.359 7 6</td>
</tr>
</tbody>
</table>
Algorithm \textsc{Lnfg2Gba} Transforming an LNFG into a GBA.

\begin{itemize}
  \item \textbf{Procedure} \textsc{Lnfg2Gba}(G)
  \item \textsc{// precondition: } \( G = (CL(P), EL(P), V_0, L = [L_1, \ldots, L_m]) \) is the LNFG of a PPTL formula \( P \)
  \item \textsc{// postcondition: } \textsc{Lnfg2Gba}(G) gives the corresponding GBA \( G_b = (Q, \Sigma, \Delta, I, T) \)
  \item \textsc{begin function}
  \begin{enumerate}
    \item \( Q = \emptyset \); \( S_p = \{ i | r \) is the atomic proposition appearing in PPTL formula \( P \) \}
    \item \( S_p = \{ i | r \) or \( \neg r \) for each \( r \in S \); \( \Sigma = \{ \bigwedge_{p \in M} p | M \subseteq S_p \} \cup \{ \epsilon \} \)
    \item \( T = \{ T_1, \ldots, T_m \} \); \( T_1 = \emptyset \); \( 1 \leq i \leq m \); \( I = \emptyset \);
    \item \textsc{for each } \( v \in \text{CL}(P) \) \textsc{do}
      \begin{enumerate}
        \item \( Q = \emptyset \)
        \item \textsc{if } \( v \in V_1 \) \textsc{then}
          \begin{enumerate}
            \item \( \Delta = \Delta \cup \{ (v, \epsilon, v) \} \)
            \item \textsc{end if}
          \end{enumerate}
        \item \textsc{end for}
      \end{enumerate}
    \item \textsc{end function}
  \end{enumerate}
  \item \textsc{return } \( G_b = (Q, \Sigma, \Delta, I, T) \);
  \item \textsc{end function}
\end{itemize}

Let \( \mathcal{G}_P = (CL(P), EL(P), V_0, L = [L_1, \ldots, L_m]) \) be the LNFG of a PPTL formula \( P \), and \( S_p \) the set of all atomic propositions appearing in \( P \). Further, \( S_p = \{ \neg r | r \in S \} \), \( S_p = S_p \cup S_p \), and \( \Sigma = \{ \bigwedge_{p \in M} p | M \subseteq S_p \} \). In \( \mathcal{G}_P \), a node \( v \in \text{CL}(P) \) denotes a PPTL formula while an edge from node \( v_l \) to \( v_j \) is a tuple \((v_l, Q_e, v_j)\) where \( v_l \) and \( v_j \) are PPTL formulas and \( Q_e \in \Sigma \). Now we define the accepting words of the LNFG \( \mathcal{G}_P \) on \( \Sigma \).

An infinite word \( u \) over \( \Sigma \) is an infinite sequence \( u = u_0u_1 \ldots \), with \( u_i \in \Sigma \) for each \( i \in N_0 \), and a finite word \( u' \) over \( \Sigma \) is a finite sequence \( u' = u'_0u'_1 \ldots u'_n \) with \( u'_i \in \Sigma \) for each \( 0 \leq i \leq n \). An infinite run \( \sigma \) of \( \mathcal{G}_P \) on an infinite word \( u = u_0u_1 \ldots \) is an infinite sequence \( q_0q_1 \ldots \) where \( q_0 \in V_0 \) and for every \( i \in N_0 \), \( q_i \in \text{CL}(P) \) and \( (q_i, u_i, q_{i+1}) \in \text{EL}(P) \). \( \sigma \) is an accepting run if all the nodes appearing infinitely often in \( \sigma \) have no common fin labels. A finite run \( \sigma' \) of \( \mathcal{G}_P \) on a finite word \( u' = u'_0u'_1 \ldots u'_n \) is a finite sequence \( q'_0q'_1\ldots q'_{n+1} \) where \( q'_0 \in V_0 \), \( q'_{n+1} \in \text{CL}(P) \), for every \( i \) \((0 \leq i \leq n) \), \( q'_i \in \text{CL}(P) \) and \( (q'_i, u', q'_{i+1}) \in \text{EL}(P) \). \( \sigma' \) is an accepting run if \( q'_{n+1} = \epsilon = 1 \).

An infinite word \( u \) over \( \Sigma \) is an infinite sequence \( u = u_0u_1 \ldots \), with \( u_i \in \Sigma \) for each \( i \in N_0 \). A run \( \sigma \) of \( \mathcal{G}_A \) on an infinite word \( u = u_0u_1 \ldots \) is an infinite sequence of states \( q_0q_1 \ldots \) where \( q_0 \in I \) and for every \( i \in N_0 \), \( q_i \in Q \) and \( (q_i, u_i, q_{i+1}) \in \Delta \). A run \( \sigma \) is an accepting run if for each \( j \) \((1 \leq j \leq r) \), it uses infinitely many transitions from \( T_j \). The language of \( \mathcal{G}_A \), \( L(\mathcal{G}_A) \), is the set of words on which there exists an accepting run \( \sigma \) of \( \mathcal{G}_A \).

Definition 3. A generalized Büchi automaton [12] over \( \Sigma \) is a five-tuple \( \mathcal{G}_A = (Q, \Sigma, \Delta, I, T) \), where \( Q \) is a finite, non-empty set of states, \( \Sigma = \{ \bigwedge_{p \in M} p | M \subseteq S_p \} \) the alphabet, \( \Delta \subseteq Q \times \Sigma \times Q \) the transition relation, \( I \subseteq Q \) the set of initial states, and \( T = \{ T_1, \ldots, T_m \} \) the acceptance condition where for each \( j \), \( 1 \leq j \leq r \), \( T_j \subseteq \Delta \).

An infinite word \( u \) over \( \Sigma \) is an infinite sequence \( u = u_0u_1 \ldots \), with \( u_i \in \Sigma \) for each \( i \in N_0 \). A run \( \sigma \) of \( \mathcal{G}_A \) on an infinite word \( u = u_0u_1 \ldots \) is an infinite sequence of states \( q_0q_1 \ldots \) where \( q_0 \in I \) and for every \( i \in N_0 \), \( q_i \in Q \) and \( (q_i, u_i, q_{i+1}) \in \Delta \). A run \( \sigma \) is an accepting run if for each \( j \) \((1 \leq j \leq r) \), it uses infinitely many transitions from \( T_j \). The language of \( \mathcal{G}_A \), \( L(\mathcal{G}_A) \), is the set of words on which there exists an accepting run \( \sigma \) of \( \mathcal{G}_A \).

Similar to the approach adopted in SPIN for modeling finite behaviors of a system with a Büchi automaton, the stuttering rule [16] is adopted so that the classic notion of acceptance for finite runs (thus words) would be included as a special case in GBA and BA. To apply the rule, we extend the alphabet \( \Sigma \) with a fixed predefined null-label \( \epsilon \), representing a no-op operation that is always executable and has no effect.

Now we present the transformation from an LNFG to a GBA. Let \( \mathcal{G}_P = (CL(P), EL(P), V_0, L = [L_1, \ldots, L_m]) \) be the LNFG of a PPTL formula \( P \), \( S_P \) the set of all atomic propositions appearing in \( P \), \( S_P = \{ \neg r | r \in S_P \} \), and \( S_P = S_P \cup S_P \). A GBA \( \mathcal{G}_A = (Q, \Sigma, \Delta, I, T) \) can be obtained as follows:

- \( Q = \text{CL}(P) \)
- \( \Sigma = \{ \bigwedge_{p \in M} p | M \subseteq S_p \} \cup \{ \epsilon \} \)
- \( I = V_0 \)
- \( \Delta = \{ (v, \alpha, v') | v = \epsilon, v' = \epsilon \} \) or \( \{ (v, \alpha, v') | v, v' \in \text{CL}(P) \} \)
- \( T = \{ T_i | i \in L_1, 1 \leq i \leq m \} \), where \( T_i = \{ (v, \alpha, v') | (v, \alpha, v') \in \Delta \) and \( v' \notin L_i \} \).

According to the above transformation, Algorithm \textsc{Lnfg2Gba} is formalized for transforming an LNFG to a GBA. In Algorithm \textsc{Lnfg2Gba}, there are two nested loops: the first one iterates through all the transitions in \( \Delta \), and the second one iterates through the label set \( L \).
Theorem 4. Let \( G_P = (V_0,\text{CL}(P),\text{EL}(P),\mathbb{L} = \{L_1,\ldots,L_m\}) \) be an LNFG of a PPTL formula \( P \), and \( G_A = (Q,\Sigma,\Delta,I,T = (T_1,\ldots,T_m)) \) be a GBA generated from \( G_P \) by Algorithm LNFG2GBA. We have \( L(G_A) = L(G_P) \).

Proof. First we prove \( L(G_A) \subseteq L(G_P) \). There are two cases we need to consider: (1) Let \( \sigma = q_0q_1 \ldots \) be an accepting run of \( G_A \) on a word \( u = u_0u_1 \ldots \) and there is no \( \varepsilon \) node in \( \sigma \). Hence, \( q_0 \in I \), and for every \( i \in N_0 \), \( q_i \in Q \) and \( (q_i,u_i,q_{i+1}) \in \Delta \). According to Algorithm LNFG2GBA, we can build a run \( \sigma' = q_0'q_1' \ldots \) of \( G_P \) such that \( q_0' \in V_0 \) and for all \( i \geq 0 \), \( q_i' = q_i \) and \( (q_i,u_i,q_{i+1}) \in \text{EL}(P) \). Now we assume that \( \sigma' \) is not an accepting sequence of \( G_P \). According to the accepting condition of an LNFG, all of the nodes appearing infinitely often in \( \sigma' \) have common labels. Suppose \( q_1' \ldots q_n' \) are the nodes appearing infinitely often in \( \sigma' \) and the common label is \( \text{fin}(q_i) \) where \( 1 \leq k \leq m \). Hence, for any \( j (1 \leq j \leq n) \), we obtain

\[
\begin{array}{l}
(q_{j-1},u_{j-1},q_j) \notin T_k & \text{if } i < j \leq n \text{ and } (q_{j-1},u_{j-1},q_j) \in \Delta \\
(q_n,u_n,q_j) \notin T_k & \text{if } j = 1 \text{ and } (q_n,u_n,q_i) \in \Delta
\end{array}
\]

Thus, there is no transition from \( T_k \) appearing infinitely many times in \( \sigma' \), and this contradicts the accepting condition of a GBA. Therefore, \( \sigma' \) is an accepting run of LNFG \( G_P \). (2) Let \( \sigma = q_0q_1 \ldots \) be an accepting run of \( G_A \) on a word \( u = u_0u_1 \ldots \) and there is \( \varepsilon \) node in \( \sigma \) and we assume it is \( q_0 \), \( n \in N_0 \). Then we have, \( q_0 \in I \), for every \( i \) \((0 \leq i < n)\), \( q_i \in Q \) and \( (q_i,u_i,q_{i+1}) \in \Delta \) and for \( i \geq n \), \( q_i \) is \( \varepsilon \) and \( (q_i, \varepsilon, q_{i+1}) \in \Delta \). Since \( \varepsilon \) represents a no-op operation, according to Algorithm LNFG2GBA, we can build a finite run \( \sigma' = q_0'q_1' \ldots q_n' \) of \( G_P \) on \( u = u_0u_1 \ldots u_{n-1} \) with \( q_0' = q_0 \), \( q_1' = q_1 \). Since there is \( \varepsilon \) node \( q_n' \) in \( \sigma' \), \( \sigma' \) is an accepting run of \( G_P \).

Now, we prove \( L(G_P) \subseteq L(G_A) \). There are also two cases we need to consider: (1) Let \( \sigma' = q_0'q_1' \ldots \) be an infinite accepting run of \( G_P \) on an infinite word \( u = u_0u_1 \ldots \). Similarly, we can also build a run \( \sigma = q_0q_1 \ldots \) of \( G_A \) with \( q_0 = q_0' \) and \( q_1 = q_1' \). Since \( \sigma' \) is an infinite accepting run of \( G_P \), according to the accepting condition of an LNFG, all of nodes appearing infinitely many times in \( \sigma' \) have no common labels. Suppose \( q_1' \ldots q_n' \) are all the nodes appearing infinitely often in \( \sigma' \). This means that \( q_1 \ldots q_n \) have no common labels. For the label set \( \mathbb{L} = \{L_1,\ldots,L_m\} \), it has that for each \( k(1 \leq k \leq m) \), there exists \( j (1 \leq j \leq n) \) such that \( q_j \) is not labeled by \( \text{fin}(l_k) \), and this infers

\[
\begin{array}{l}
(q_{j-1},u_{j-1},q_j) \in T_k & \text{if } i < j \leq n \text{ and } (q_{j-1},u_{j-1},q_j) \in \Delta \\
(q_n,u_n,q_j) \in T_k & \text{if } j = 1 \text{ and } (q_n,u_n,q_i) \in \Delta
\end{array}
\]

Hence, for each \( k(1 \leq k \leq m) \), \( \sigma \) uses infinitely many transitions from \( T_k \). According to the accepting condition of a GBA, \( \sigma \) is an accepting run. (2) Let \( \sigma' = q_0'q_1' \ldots q_n' \) be a finite accepting run of \( G_P \) on a word \( u = u_0u_1 \ldots u_{n-1} \) with \( q_0' \in V_0 \), \( q_n' = \varepsilon \) and for each \( i (0 \leq i < n) \), \( (q_i,u_i,q_{i+1}) \in \text{EL}(P) \). According to Algorithm LNFG2GBA, we can build an infinite run \( \sigma = q_0q_1 \ldots q_nq_{n+1} \ldots \) of \( G_A \) with \( q_0 \in I \), for \( 0 \leq i < n \), \( q_i = q'_i \) and \( (q_i,u_i,q_{i+1}) \in \Delta \) and for \( i \geq n \), \( q_i = \varepsilon \) and \( (q_i, \varepsilon, q_{i+1}) \in \Delta \). Since \( \varepsilon \) is not labeled by \( \text{fin} \), for each \( k (1 \leq k \leq m) \), \( (\varepsilon, \varepsilon, \varepsilon) \in T_k \). Hence, for each \( k(1 \leq k \leq m) \), \( \sigma \) uses infinitely many transitions from \( T_k \). According to the accepting condition of a GBA, \( \sigma \) is an accepting run. \( \square \)

There are several ways to transform a GBA to a BA. We adopt a way presented in [19] in our PPTL model checking approach.

Analysis of the complexity: It has been proved in [20] that the time complexity of the decision procedure is inherently non-elementary. Essentially, the non-elementary complexity of PPTL is caused by chop constructs, since negations cannot be pushed in front of atomic propositions for a formula with chop operators. Specifically, for a given PPTL formula, the complexity for checking the satisfiability will be \( n \)-exponential, where \( n \) is the nesting depth of chop and negation operations in the formula. So, when the nesting depth of chop and negation operations is more than \( 1 \), the complexity will be exponential. In practice, the nesting depth of chop and negation operations of a formula used for specifying the desired property of a program is no more than \( 1 \). So the complexity is really exponential for model checking.

5.2. Model checking example

A translator has been developed in C++ for realizing the transformation from a PPTL formula to a Büchi automaton using the algorithms presented in this paper. It has been successfully integrated in the model checker SPIN [15]. In the following, we use an example borrowed from [18] to show how it works.

Traffic light control system (TLCs) is common in our daily life. As we all know, the duration of the green lights for the main road should be longer than that of the red lights in the rush hours. Now a simple rule is made for TLCs. We assume there are two modes in the system. Mode 1 represents the rush hours and mode 0 represents the other time. When the current time is between 7 o’clock and 9 o’clock or between 17 o’clock and 19 o’clock, the system is in mode 1. The system can translate from mode 0 to mode 1 and from mode 1 to mode 0 according to time o’clock. The details about how TLCs works are presented below.

(1) The system starts at 0 o’clock;
(2) The mode of the system is set as 0. The green light of the east–west direction and the red light of the south–north direction are on. This state lasts 25 seconds;
(3) The yellow light of the east–west direction flashes and the red light of the south–north direction is on. This state lasts 5 seconds;
(4) The red light of the east–west direction and the green light of the south–north direction are on. This state lasts 25 seconds;

(5) The red light of the east–west direction is on and the yellow light of the south–north direction flashes. This state lasts 5 seconds. According to the current time, the mode is set for the next state. If the current time is between 7 o’clock and 9 o’clock or between 17 o’clock and 19 o’clock, the next step is (6), otherwise the next step is (2);

(6) The mode of the system is set as 1. The green light of the east–west direction and the red light of the south–north direction are on. This state lasts 30 seconds;

(7) The yellow light of the east–west direction flashes and the red light of the south–north direction is on. This state lasts 5 seconds;

(8) The red light of the east–west direction and the green light of the south–north direction are on. This state lasts 20 seconds;

(9) The red light of the east–west direction is on and the yellow light of the south–north direction flashes. This state lasts 5 seconds. According to the current time, the mode is set for the next state. If the current time is between 7 o’clock and 9 o’clock or between 17 o’clock and 19 o’clock, the next step is (6), otherwise the next step is (2).

Here we use EW_G, EW_Y, EW_R, SN_G, SN_Y and SN_R to represent the green light of the east–west, the yellow light of the east–west, the red light of the east–west, the green light of the south–north, the yellow light of the south–north or the red light of the south–north direction is on respectively.

We model the TLCS by PROMELA [15], the modelling language of SPIN, as shown in Fig. 6. In the program, the east–west lights in red, yellow and green are denoted by EW_RED, EW_YELLOW, and EW_GREEN respectively, and the south–north lights in red, yellow and green are denoted by SN_RED, SN_YELLOW, and SN_GREEN respectively. We use boolean variable sign to denote the mode and int variable t to denote the time. If the current time is between 7 o’clock and 9 o’clock or between 17 o’clock and 19 o’clock, the value of sign is true, that is to say it is in mode 1.

First we consider a safety property: at any state, the yellow or green light of the south–north direction works when the red light of the east–west direction is on, and the yellow or green light of the east–west direction works when the red light of the south–north direction is on. We can specify this property by a PPTL formula as follows:

\[ P \equiv \Box (p \land r \lor q \land r \lor s \land v \lor t \land v) \]

where

\#define p EW_GREEN == true

\#define q EW_YELLOW == true
Using SPIN model checker for PPTL, the verification result shown in Fig. 7 is reported. As we can see, the property is valid.

Now we check a periodically repeated property in mode 0: every other state in which the green light of the east–west direction and the red light of the south–north direction are on. The property is not required to be satisfied on the TLCS model. This property cannot be described by any LTL formula. Certainly, we can specify this property by a PPTL formula as follows:

\[ P = (p \rightarrow (q \land r)); ((\Box^2 (p \rightarrow (q \land r)))^* \]

where

# define p sign == false
# define q EW_GREEN == true
# define r SN_RED == true

Using SPIN model checker for PPTL, the result in Fig. 8 is provided. As we can see, the verification result is not valid. Thus, a counterexample is given in Fig. 9.

6. Conclusion

A canonical form for chop formulas is defined in this paper. It has been used to simplify the constructing algorithm of LNFGs for PPTL formulas. Further, an improved decision procedure and model checking approach based on SPIN for PPTL are formalized. However, the time complexity of the algorithms is high as full regular properties are checked. Therefore, in the future, we need further to improve the model checking approach with PPTL and fight with the state space explosion problem. For doing so, we will go two ways by using our improved new constructing algorithm of LNFGs for PPTL formulas. On one direction, we will further improve traditional model checking approaches such as symbolic model checking (SMC) [21], bounded model checking (BMC) [22], and abstract model checking (AMC) [23,24] for PPTL in order to verify full regular properties. With these approaches, the model of a system is specified by a Kripke structure or automaton while the property
to be verified is specified by a PPTL formula. On the other direction, we will further improve our unified model checking (UMC) approach [25]. With this method, the model of a system is described by an MSVL program while the property to be verified is defined in a PPTL formula or MSVL program. Then, we will develop a unified AMC (UAMC) by means of our improved CEGAR approach [26,27]. Further, we will improve the UAMC approach based on dynamic symbolic execution (DSE) which can be a complement to both CEGAR and UAMC approaches [28].
References


