Compositional Verification with Stutter-invariant Propositional Projection Temporal Logic *

Chen Yang Zhenhua Duan
Institute of Computing Theory and Technology
Xidian University, Xi’an 710071, People’s Republic of China
{chyang,zhhduan}@mail.xidian.edu.cn

Abstract: This paper investigates compositional verification with Propositional Projection Temporal Logic (PPTL). To this end, a sublogic Stutter-invariant PPTL (written as PPTL$_{st}$) is first proposed. As a specification language, PPTL$_{st}$ helps designers get rid of irrelevant detail in compositional verification of a concurrent system since the projection construct of PPTL$_{st}$ allows designers to assert formulas over points of interest through an execution. In this way, modules can be abstracted based on their local properties defined by projection construct, and substitution of modules by the abstracted ones will not affect overall properties of the system. PPTL$_{st}$ is proved to be able to capture all stutter-invariant properties expressive in PPTL. Further, an algorithm translating PPTL formulas to PPTL$_{st}$ ones is also given. And the complexity of the algorithm is accordingly studied. Moreover, an example—automatic gas station, is studied to illustrate compositional verification with PPTL$_{st}$.

Key–Words: Propositional projection temporal logic, Stutter-invariance, Compositional verification, Partial-order model-checking

1 Introduction

Lamport has argued that a specification of concurrent systems should be stutter-invariant [9, 18]. A specification that is not invariant under stuttering will not allow refinement or abstraction and thus will be useless in hierarchical (modular) verification. Besides, experience has shown that the best way to compositionally verify concurrent systems is through a hierarchy of levels of abstraction, starting from abstraction of each module by considering only relevant state-changes so that the local properties are not affected, and ending with proof of overall property in the reduced system formed by replacing each module by its abstraction. Propositional Linear Temporal Logic (PLTL)[17], is a popular alternative for specifying properties in the formal verification of concurrent systems. One way to assure that only stutter-invariant properties are specified is to restrict PLTL to only those formulas without “next” operator. Peled proved that “next”-free PLTL can accept all stutter-invariant properties expressible in PLTL[9].

However, PLTL offers a poor support for compositional (modular) verification since it is not capable of supporting abstraction. It has been mentioned by Mokkedem that semantics of PLTL is defined in terms of global states in such a way that temporal properties of a given component, viewed within some context, do not make abstraction of invisible state-changes by other components [1]. Here the “invisible state-change” is equal to the transition that does not change values of propositional variables included in the property formula.

Propositional Projection Temporal Logic (PPTL) [24, 25], as an extension of Propositional Interval Temporal Logic (PITL) [3], provides a new projection construct $(P_1, \ldots, P_m) \mathsf{prj} Q$ (symbol “$\mathsf{prj}$ ” denotes projection operator) for modular abstraction. This enables PPTL to support compositional verification (The comparison between the original projection construct of PITL and the new one can refer to [22]). Intuitively, as shown in Fig. 1, the projection construct means that formula $Q$ is interpreted in parallel with sequential execution $P_1; \ldots; P_m$ (symbol “$;$” denotes chop operator) over an interval obtained by taking the endpoints (rendezvous points) of the intervals over which formulas $P_1, \ldots, P_m$ are interpreted. Significantly, when the projection construct is used to specify a concurrent system $S$, the left hand side, $(P_1, \ldots, P_m)$, of the construct can be employed to abstract executions of $S$ by restricting each execution over points of interest (i.e. restricting point-sequence $s_0, \ldots, s_n$ over $s'_0, \ldots, s'_4$ in the figure), while the right hand side $Q$ can be used to express (safety or liveness) properties of the abstraction.

For example, the payment transaction via a pos
machine can be modeled by the communication of two individual processes \textit{pos\_machine} and \textit{account} in ProMeLa language [10], which simulate the pos machine and bank account management respectively, as shown in Fig. 2(1). For better understanding, let’s have a brief look at ProMeLa. The \textit{do} statement is repetition structure. Various options are specified after doubled colons “::”, and only one option can be selected for execution at a time. The \textit{if} statement is represented very similarly to the \textit{do} except for repetition. Message passing is represented in CSP notation [4]. For instance, \textit{c!m} indicates that the message (data) \textit{m} should be sent along channel \textit{c}. Similarly, \textit{c?m} indicates that a message received on channel \textit{c} should be stored in variable \textit{m}. Back to our example, with parallel composition [4] (denoted by symbol “||”) of the two processes, we obtain an automaton, as shown in the middle of the Fig. 2(1), which includes all possible executions. In the automaton, we use \textit{S} to denote the synchronization of \textit{c!n} and \textit{c?m}, i.e. \textit{S_c = c!n || c?m}. And on each execution, we will check if the following PPTL property can be faithfully satisfied

\[ \varphi = (e_s, e_a, e_r, e_p) \text{ prj } (((p_1; q_1) \lor (p_2; q_2)) \]

where proposition \textit{e_s}, \textit{e_a}, \textit{e_r}, and \textit{e_p} respectively represent occurrences of transitions swipe, \textit{S\_access}, \textit{S\_reply} and prepay in the automaton, and \textit{p_i} is defined as equation \textit{usr = cus_i}, \textit{q_i} as \textit{payer = cus_i}. The chop construct \textit{p ; q} indicates proposition \textit{q} eventually becomes true if \textit{p} holds now. In this way, the property says that (i) along each execution which is a sequential composition of only four transitions: \textit{e_s, e_a, e_r} and \textit{e_p}, (ii) customer \textit{cus_i} can pay money only from his account (\textit{payer = cus_i}).

Observe that there are only two executions in the automaton, both of which begin from transition (\textit{swipe? usr, pas, am}) and end in (\textit{prepay! payer, am}). They distinguish each other by paths indicated by dotted lines and circles, whereas the sequence of (i) can be obtained from either of the two executions. Meanwhile, \textit{(p_1 ; q_1)} holds in one execution and \textit{(p_2 ; q_2)} in the other. Thus, the property holds in the automaton. Moreover, the automaton can be abstracted by ignoring the dotted paths, so from the view of its environment, the automaton becomes a more succinct one, as shown in the right of Fig. 2(1). In Section 4, we will study the compositional verification of a concurrent system—Automatic Gas Station, in which the pos system is regarded as a module. To avoid considering irrelevant detail, we use the abstracted automaton in stead of the original one for the pos system in the verification.

Therefore, in this way projection formula \textit{\varphi} can guide abstracting component \textit{\kappa} which it specifies. In particular, invisible transitions of \textit{\kappa} that cause exponential increase in size of parallel composition with other components are abstracted away. Nevertheless, to guarantee the abstraction being equivalent to \textit{\kappa} w.r.t \textit{\varphi} and also satisfying \textit{\varphi}, formula \textit{\varphi} itself must be stutter-invariant. So, we are motivated to propose stutter-invariant PPTL in order to support compositional verification.

The decision procedure for PPTL given in [5] can be applied in explicitly model-checking [6]. Besides, it is proved that PPTL can express exactly the full regular language (\omega-regular + regular languages) [7]. This also encourages us to investigate stutter-invariant PPTL for compositional verification. To this end, we first examine all constructs of PPTL, which can generate only stutter-invariant formulas. We call the sub-logic restricted by these constructs stutter-invariant PPTL, denoted by PPTL\textsubscript{\textit{st}}. Then, we prove that PPTL\textsubscript{\textit{st}} can express every stutter-invariant property expressible in PPTL. Following the proof, we propose a translation function \textit{\tau} of PPTL formula \textit{\varphi} into its corresponding PPTL\textsubscript{\textit{st}} formula \textit{\tau(\varphi)}. Thus, checking whether a PPTL formula defines a stutter-invariant property is induced to checking the validity of \textit{\varphi} \leftrightarrow \textit{\tau(\varphi)}. Finally, we show that this translation incurs blowup with an upper bound of \(2^{O(n \log k)} \times |\varphi|\) in a worst case, where \textit{n} is the number of distinct propositions in \textit{\varphi}, and \textit{k} is the “next”-depth of \textit{\varphi}.

Besides, partial-order model-checking against PPTL\textsubscript{\textit{st}} properties is the underlying technique to support our compositional verification. The technique is based on automata-theoretic approach. More precisely, PPTL\textsubscript{\textit{st}} formulas are employed to define the properties to be checked, and the negation of each formula is transformed to a Büchi automaton \textit{A_p}. The transformation algorithm can be found in [6, 5, 7]. Meanwhile the system modeled in ProMeLa language [10] is translated to an automata \textit{A_s} as well. Then, whether the system satisfies PPTL\textsubscript{\textit{st}} properties or not can be checked by computing the product automaton of \textit{A_s} and \textit{A_p}, and checking whether or not the product automaton accepts the empty word. The above method has been implemented based on the successful model checker SPIN [10, 11] so that the partial-order model-checking algorithm included in SPIN can be used off-the-shelf.

The rest of the paper is organized as follows. In the following section, we briefly introduce PPTL. In Section 3, PPTL\textsubscript{\textit{st}} is introduced based on the defini-
tion of stutter-invariance w.r.t PPTL. In addition, the method of determining stutter-invariance of PPTL formulas is proposed, and its complexity is accordingly studied. In Section 4, we give an example to illustrate compositional verification in PPTL. Finally, we draw the conclusion in the last section.

Related work Lamport’s Temporal Logic of Actions (TLA) [19] is the first stutter-invariant logic to specify programs. Together with Mokkedem’s Modular Temporal Logic (MTL) which is close under ω-stuttering [1], their semantics preserve the truth of formulas specifying modules under parallel composition so that compositional reasoning is supported. Nevertheless, both of them appeal to a set of program’s variables that must be provided. As succinct as PLTL, the semantics of PPTL does not depend on these variables. Considering fairness, Alexander posed a problem that it cannot be detected whether or not an action is treated fair in stutter-equivalent executions [2]. He adapted TLA’s semantic model based on partial-order to cope with this problem, and proposed Temporal Logic of Distributed Actions (TLDA) for compositional design of systems. In fact, fairness can be determined by PPTL since occurrences of actions are allowed to be directly specified with projection construct (see the above example). For the power of expressiveness, Etessami proposed Stutter-invariant Existential Quantified Linear Propositional Temporal Logic (SI-EQLTL) which precisely expresses ω-regular language [15], whereas PPTL can express full-regular (regular + ω-regular) language since formulas are interpreted over both finite and infinite intervals [7]. In addition, it seems that both Monadic Second-order Logic (MSO) [13] and PPTL allow to assert temporal property over points of interest in an execution. However MSO depends on quantifiers. With the classical definition, quantification does not make abstraction to stuttering [21]. And specifications with MSO or SI-EQLTL suffer from the proliferation of extra time variables and quantifiers. PPTL is as complex as MSO, but as a pure propositional logic, it could be more easily applied in specification of concurrent systems.

2 Propositional Projection Temporal Logic

This section briefly introduces Propositional Projection Temporal Logic (PPTL). The details of the logic can be found in [24, 5, 25].

2.1 Syntax

Let \( AP \) be a non-empty set of atomic propositions. The formula of PPTL is given by the following grammar:

\[
P := p | \neg P | P_1 \lor P_2 | \Box P | P_1; P_2 | (P_1, ..., P_m) \text{prj} Q
\]

where \( p \in AP \). \( P_1, \ldots, P_m \), \( P \) and \( Q \) are all well-formed PPTL formulas. “;”(chop), “□” (next) and “prj” (projection) are basic temporal operators. A formula is called a state formula if it does not contain any temporal operators (“;”, “prj ” and “□”) otherwise it is a temporal formula.

2.2 Semantics

States Since PPTL is interpreted over intervals composed of consecutive states, we define a state \( s \) over \( AP \) to be a mapping from \( AP \) to \( B = \{true, false\} \):

\[
s : AP \rightarrow B
\]

We use \( s(p) \) to denote the valuation of \( p \) at the state \( s \).

Intervals 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. To have a uniform notation for both finite and infinite intervals, we use extended integers as indices. That is, we define the extended integers as
a union of the set $N_0$ of non-negative integers and $\{\omega\}$, $N_\omega = N_0 \cup \{\omega\}$, and extend the comparison operators $=$, $<$, and $\leq$ to $N_\omega$ by considering $\omega = \omega$ and for all $i \in N_0$, $i < \omega$. Furthermore, we define $\leq$ as $\leq - \{(\omega, \omega)\}$.

To simplify the definitions, we denote $\sigma$ as $<s_0, \ldots, s_{|\sigma|}>$ where $s_{|\sigma|}$ is undefined if $\sigma$ is infinite. With such a notation, $\sigma(i, \ldots, j)(0 \leq i \leq j \leq |\sigma|)$ denotes the sub-interval $<s_i, \ldots, s_j>$ and $\sigma(k)(0 \leq k \leq |\sigma|)$ denotes $<s_k, \ldots, s_{|\sigma|}>$. The concatenation of a finite $\sigma$ with another interval (or empty string) $\sigma'$ is denoted by $\sigma \cdot \sigma'$.

Let $\sigma = <s_0, \ldots, s_{|\sigma|}>$ be an interval and $r_1, \ldots, r_n$ be integers ($n \geq 1$) such that $0 \leq r_i \leq \ldots \leq r_n \leq |\sigma|$. The projection of $\sigma$ onto $r_1, \ldots, r_n$ is the interval $\sigma \downarrow (r_1, \ldots, r_n) = <s_{r_0}, s_{r_1}, \ldots, s_{r_n}>$ where $s_0, t_1, \ldots, t_l$ is obtained from $r_1, \ldots, r_n$ by deleting all duplicates. That is, $t_1, \ldots, t_l$ is the longest strictly increasing subsequence of $r_1, \ldots, r_n$. For instance,

$<s_0, s_2, s_3 >^\downarrow (0, 0, 2, 2, 3) = <s_0, s_2, s_3>$

Thus, it is convenient for us to define an interval obtained by taking the endpoints (rendezvous points) of the intervals over which $P_1, \ldots, P_m$ are interpreted in the projection construct.

**Interpretations** An interpretation is a tuple $I = (\sigma, k, j)$, where $\sigma$ is an interval, $k$ integer, and $j$ an integer or $\omega$ such that $k \leq j \leq |\sigma|$. We use the notation $(\sigma, k, j) \models P$ to mean that some formula $P$ is interpreted and satisfied in the sub-interval $<s_k, \ldots, s_j>$ of $\sigma$ with the current state being $s_k$.

The satisfaction relation ($|)$ is inductively defined as follows:

$-I \models p$ iff $s_I(p) = true$, for any given proposition $p$.

$-I \models \neg P$ iff $I \not\models P$.

$-I \models P \lor Q$ iff $I \models P$ or $I \models Q$.

$-I \models (P_1 \cdot P_2)$ iff there exists $r$ such that $k \leq r \leq j$, $(\sigma, k, r) \models P_1$ and $(\sigma, r, j) \models P_2$.

$-I \models \circ P$ iff $k < j$ and $(\sigma, k + 1, j) \models P$.

$-I \models (P_1, \ldots, P_m) \circ P$ iff there exist integers $k = r_0 \leq r_1 \leq \ldots \leq r_m$ such that $(\sigma, r_{l-1}, r_l) \models P_l$ for $(1 \leq l \leq m)$, and $(\sigma', 0, |\sigma'|) \models Q$ for one of the following:

(i) $r_m < j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_m) \cdot \sigma(r_{m+1}, \ldots, j)$ or

(ii) $r_m = j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_m)$

The following are some useful derived formulas ($n \geq 1, n \in N_0$).

$false \overset{def}{=} P \land \neg P$ 

$e \overset{def}{=} \neg \circ true$

$\diamond P \overset{def}{=} true; P \overset{def}{=} \circ P$

$\square P \overset{def}{=} \neg \diamond \neg P$

$\diamond^n P \overset{def}{=} \circ (\circ^{n-1} P)$

where $\diamond$ (sometimes) and $\square$ (always) are derived temporal operators; $e$ (empty) denotes an interval with zero length.

### 3 Stutter-Invariance of PPTL

#### 3.1 Stutter-invariance

Since PPTL formulas are interpreted over intervals, we first consider the definition of stuttering w.r.t. intervals below.

**Definition 1 (Stuttering)** Stuttering occurs in an interval $\sigma$ of states when a state occurs two or more times consecutively; that is, for $\sigma = <s_0, \ldots, s_{|\sigma|}>$ if there is some $i \in N$ such that $s_i = s_{i+1}$ so that $\sigma = <s_0, \ldots, s_{i}, s_{i+1}, s_{i+2}, \ldots, s_{|\sigma|}>$.

In [9], Peled gives the definition of stutter-equivalent linear-time structures for PLTL. We adopt the analogous definition for PPTL based on stuttering intervals.

**Definition 2 (Stutter-equivalence)** Two intervals $\sigma = <s_0, \ldots, s_{|\sigma|}>$ and $\sigma' = <s'_0, \ldots, s'_{|\sigma'|}>$ are said to be stutter-equivalent, written as $\sigma \sim_{st} \sigma'$, if there are infinite or finite sequences $0 = i_0 < i_1 < i_2 < \ldots$ and $0 = j_0 < j_1 < j_2 < \ldots$ such that for every $k \geq 0$:

$s_{i_k} = s_{i_{k+1}} = \ldots = s_{i_{k+1}}$.

$s'_{j_k} = s'_{j_{k+1}} = \ldots = s'_{j_{k+1}}$

As shown in Fig. 3, intervals $\sigma$ and $\sigma'$ are stutter-equivalent, and they have the same trend in state-change.

$\sigma : s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5$

$\sigma' : s_0' = s_0 \rightarrow s_1' \rightarrow s_2' = s_2 \rightarrow s_3' \rightarrow s_4' = s_4 \rightarrow s_5'$

Figure 3: An example of stutter-equivalent intervals

A property is a set of intervals interpreted over a set $AP$ of atomic propositions. On the basis of stutter-equivalent intervals, we present an analogous definition of stutter-invariant property in order to introduce properties expressible in PPTL as follows:

**Definition 3 (Stutter-invariant property)** A property is stutter-invariant if the set of intervals which characterizes the property is a union of stutter-equivalence classes. That is, whenever $\sigma$ and $\sigma'$ are stutter-equivalent intervals then either $\sigma$ and $\sigma'$ belong to this set or none of them belongs to this set.

**Definition 4 (Property expressible in PPTL)** A property is said to be defined by PPTL formula $\varphi$ if it is the set of all intervals $\sigma$ interpreted over $AP$ such that $(\sigma, 0, |\sigma|) \models \varphi$. In this case, one also can say that this property is expressible in PPTL.

We say that an interval $\sigma = <s_0, \ldots, s_{|\sigma|}>$ is stutter-free if either $s_i \neq s_{i+1}$ for every $i \geq 0$ or there exists a $k \geq 0$ such that $s_k = s_{k+1}$ for $i < k$ and $s_k = s_{k+1} = \ldots = s_{|\sigma|}$. Note that stutter-free intervals can be finite or infinite. And every suffix of a stutter-free interval is also a stutter-free interval. Further,
each stutter-equivalence class contains stutter-free intervals, so an interval belongs to a class if it is stutter-equivalent to the stutter-free interval included in the class.

**Definition 5 (Stutter-invariant formula)** A PPTL formula \( \varphi \) is stutter-invariant if and only if for each pair of stutter-equivalent intervals \( \sigma \sim_{st} \sigma' \), we have \((\sigma, 0, |\sigma|) \models \varphi \) if and only if \((\sigma', 0, |\sigma'|) \models \varphi \).

### 3.2 Stutter-invariant PPTL (PPTL_{st})

**Lemma 1** The set of all stutter-invariant PPTL formulas is denoted by \( P_{st} \). The following are some closure properties of \( P_{st} \):

1. If \( p \in AP \), then \( p \in P_{st} \).
2. If \( \neg p \in P_{st} \), then \( \neg p \in P_{st} \).
3. If \( p, q \in P_{st} \), then \( p \land q \in P_{st} \).
4. If \( p \in P_{st} \), then \( \Box p \in P_{st} \).
5. If \( p, q \in P_{st} \), then \( p \land q \in P_{st} \).
6. If \( p_1, \ldots, p_m, q \in P_{st} \), then \(((p_1, \ldots, p_m) \mathcal{P} q) \in P_{st} \).
7. If \( p, q \in P_{st} \), then \( (\Box p ; \Box q) \lor q \in P_{st} \).

**Proof:** Suppose \( I = (\sigma, 0, |\sigma|) \), \( I' = (\sigma', 0, |\sigma'|) \) and \( I'' = (\sigma'', 0, |\sigma''|) \), we have

1. Let \( p \in AP \). \( I \models p \) if \( s_0[p] = true \). So, only the current state is relevant to the satisfaction of \( p \). As for subsequent states, stuttered or not, they have no effect on the satisfaction of \( p \) over the interval. Thus, \( p \) is stutter-invariant.

2. Let \( p \in P_{st} \). If \( \neg p \) is not stutter-invariant, there exist two stutter-equivalent intervals \( \sigma' \) and \( \sigma'' \) that \( I' \models \neg p \) and \( I'' \models p \). Note that \( I' \models \neg p \) if and only if \( I' \not\models p \). So, according to Definition 5, formula \( P \) is not stutter-invariant. This is contrary to stutter-invariance of \( P \). Therefore, \( \neg p \) is stutter-invariant.

3. Let \( p, q \in P_{st} \). If \( p \land q \) is not stutter-invariant, then there exist two stutter-equivalent intervals \( \sigma' \) and \( \sigma'' \) such that \( I' \models p \land q \). Note that \( I' \models p \land q \). Thus, in case of either \( p \) or \( q \) is not stutter-invariant, \( P \land q \) is stutter-invariant. So \( P \land q \) is stutter-invariant.

4. Let \( p \in P_{st} \). \( I \models \Box p \) if for all \( 0 \leq k \leq |\sigma| \), \((\sigma, k, |\sigma|) \models p \). As for state \( s_k \), stuttered or not, does not have effect on the satisfaction of \( p \) over the interval. So, formula \( P \) still holds over any suffix of the resulting interval. Therefore, \( \Box p \) is stutter-invariant.

5. Let \( p, q \in P_{st} \). \( I \models P ; q \) if there exists \( k \leq |\sigma| \), \((\sigma, k, |\sigma|) \models P \) and \((\sigma, k, |\sigma|) \models Q \). Because of stutter-invariance of \( P \) and \( Q \) we do not need to consider the subintervals \( \sigma(0, k) \) and \( \sigma(k, |\sigma|) \).

In this way, we just consider state \( s_k \) which cuts the interval \( \sigma \) into two sub-intervals. Note that if we replace state \( s_k \) by \( i \geq 1 \) copies of itself, the new interval we obtained, denoted by \( \sigma' \), satisfies \( \sigma \sim_{st} \sigma' \). Moreover, we have \((\sigma', 0, |\sigma'|) \models P ; Q \). So, stuttering state \( s_k \) in interval \( \sigma \) will not affect satisfaction of \( P ; Q \) on the interval. Thus, \( P ; Q \) is stutter-invariant.

### 3.3 Proof

(6) Let \( p_1, \ldots, p_m, q \in P_{st} \). Since stuttering states \( s_{k_1}, \ldots, s_{k_m} \) in interval \( \sigma \) has no effect on the fact that formula \( p_1, \ldots, p_m \) are executed sequentially over it, namely, for any interval \( \sigma' \) such that \( \sigma' \sim_{st} \sigma \) we have \((\sigma', 0, |\sigma'|) \models P_1 \ldots P_m \). Besides, since \( Q \) is stutter-invariant \((\sigma', 0, |\sigma'|) \models Q \) for one of the following:

1. \( r_m < |\sigma'| \) and \( \delta' = \sigma' \downarrow (r_0, \ldots, r_m) \).
2. \( r_m < |\sigma'| \) and \( \delta' = \sigma' \downarrow (r_0, \ldots, r_m) \).

Thus, \((P_1, \ldots, P_m) \mathcal{P} Q \) is stutter-invariant according to the semantics of projection.

(7) Let \( I \models \Box p ; \Box q \) and \( q \in P_{st} \). Therefore, \( I \models \Box p ; \Box q \). If \( I \models \Box p ; \Box q \), i.e., there exists some integer \( k \) such that \((\sigma', k + 1, |\sigma'|) \models q \) and \((\sigma', k, 0) \models \Box p \). Or \( I \models \Box p \). In the case \( s_k \neq s_{k+1} \), stuttering of any state will not affect the satisfiability of formula \( \Box p ; \Box q \).

\( \Box p \) in the resulting interval since state \( s_{k+1} \) is remained. When \( s_k = s_{k+1} \), we have \( \sigma(0, k) = \Box (p \land q) \), by stuttering, states \( s_0, \ldots, s_{k+1} \) might be merged into one state so that formula \( \Box p ; \Box q \) does not hold any more, but formula \( Q \) holds. Thus, in either case formula \( \Box p ; \Box q \) or \( Q \) is stutter-invariant.

According to Lemma 1, we define a sub-logic of PPTL (stutter-invariant PPTL, written as PPTL_{st}) as follows:

\[
P := p \mid \neg p \mid P_1 \lor P_2 \mid (P_1, \ldots, P_n) \mathcal{P} q \mid (P_1 ; \Box p_2) \lor P_2
\]

where \( p \in AP \). The next operator is allowed to appear in construct \( (\Box p_1 ; \Box P_2) \lor P_2 \).

**Lemma 2** Any property expressible in PPTL_{st} is stutter-invariant.

**Theorem 1** Any stutter-invariant property expressible in PPTL_{st} is expressible in PPTL_{st}.

Suppose \( AP = \{p_0, \ldots, p_{n-1} \} \). We will show that for every PPTL formula \( \varphi \) there exists a PPTL_{st} formula \( \tau(\varphi) \) that agrees with \( \varphi \) on all stutter-free intervals:

\[
(\sigma, 0, |\sigma|) \models \varphi \iff \tau(\varphi) \quad \text{for stutter-free } \sigma
\]

The proof is analogous to that of Theorem 1 in [9], which proceeds by induction on the structure of \( \varphi \).

**Proof:** Base For \( p \in AP \) we can simply set \( \tau(p) = p \).

**Induction** If \( \varphi \) is of the form \( \neg \varphi' \), we can set \( \tau(\varphi) = \neg \tau(\varphi') \).

Similarly, if \( \varphi \) is of the form \( \varphi' \lor \varphi'' \) or \( \varphi' ; \varphi'' \), we can set \( \tau(\varphi) = \tau(\varphi') \lor \tau(\varphi'') \) or \( \tau(\varphi) = \tau(\varphi') ; \tau(\varphi'') \) as the case may be. If \( \varphi \) is of the form \( \varphi_1, \ldots, \varphi_m \mathcal{P} \varphi' \), we can set \( \tau(\varphi) = \tau(\varphi_1) \lor \tau(\varphi_2) \lor \cdots \lor \tau(\varphi_m) \mathcal{P} \varphi' \).
(τ(φ_1),...,τ(φ_m)) pr j τ(φ'). If φ is of the form ∅φ', the situation is more difficult.

Let V be the set of all possible valuations AP → {true, false}, and for each v ∈ V, let β_v be the formula α_0 ∧ ... ∧ α_{n-1} where α_j = P_j if v(p_j) = true and α_j = ¬p_j if v(p_j) = false.

The following is the crucial observation that we will make use of: if v, v' ∈ V are such that v ≠ v', then, for any stutter-free σ over AP, we have

(σ, 0, |σ|) ∨ β_v ∨ ∅β_v' → □β_v : ∅β_v'

Therefore, (σ, 0, |σ|) ∨ ∅φ' for any stutter-free interval σ if and only if either
(a) s_0 = s_1 = ... and (σ, 0, |σ|) ∨ φ' or
(b) s_0 ≠ s_1 and σ_{1,...,|σ|} = < s_1, s_2, ..., s_{|σ|} > such that σ_{1,...,|σ|} ∨ φ'. (Recall that σ_{1,...,|σ|} is stutter-free itself.)

Thus, we can set

τ(∅φ') = \bigvee_{v ≠ v'} ((□β_v ∧ τ(φ'))) ∨ \bigvee_{v ≠ v'} ((□β_v : (β_v ∧ τ(φ')))))

Since β_v ∧ β_v' = false, formula □β_v : (β_v ∧ τ(φ')) is unsatisfiable in stutter-free interval of (a). Thus, it holds only in stutter-free interval of (b). Stuttering of interval of (b) has no effect on the satisfaction of formula □β_v : (β_v ∧ τ(φ')) in the interval, so, formula □β_v : (β_v ∧ τ(φ')) is stutter-invariant. In addition, formula (□β_v ∧ τ(φ')) is stutter-invariant. Therefore, this choice is obviously correct.

From the discussion above, we know that a PPTL formula φ is stutter-invariant if φ ↔ τ(φ).

Complexity of Determining Stutter-invariance

Suppose the number of distinct propositions appearing in φ is n. Hence the number of valuations is |V| = 2^n, i.e., there are 2^n distinct formulas β_v for each v ∈ V. Let |φ| denote the number of symbols in φ. It is not hard to derive the bounds given for |τ(φ)|. Observe that the only case which causes a blowup is a sub-formula of the form ∅φ'. From the definition of τ(∅φ'), we get |τ(∅φ')| ≤ 2^n × |τ(φ')| + 2^{O(2n)} × |τ(φ')|. Thus one can obtain as follows:

Theorem 2 Suppose τ is a translation which converts PPTL formulas to equivalent PPTL formulas. For a stutter-invariant φ, τ(∅φ) is exponentially larger than φ, with an upper-bound of |τ(∅φ)| ≤ 2^{O(nk)} × |φ| in the worst case, where n is the number of distinct propositions appearing in φ.

The correctness of τ translation follows from the fact that over stutter-free interval ∅φ is equivalent to the statement: “the first time after now that some new event happens, φ must hold, or else (if nothing new ever happens) φ must hold right now” [14].

Complexity of Satisfiability of PPTL

In [8], the lower bound for complexity of satisfiability of PPTL formulas is proved to be non-elementary. The complexity comes from negation of chop or negation of projection structure. Since PPTL inherits chop and projection operator from PPTL, so the complexity of satisfiability of PPTL formulas is also non-elementary.

As shown in Theorem 2, size of τ(φ) is exponentially larger than size of φ in the worst case. So, to check whether or not a PPTL formula defines a stutter-invariant property, one can simply check the validity of φ ↔ τ(φ). Compared with non-elementary complexity of satisfiability of PPTL, the complexity of translation τ is extremely low and thus can be ignored.

4 Case Study of Compositional Verification in PPTL_{st}

4.1 Partial-order Model-checking with PPTL_{st}

Partial-order model-checking against PPTL_{st} properties is the underlying technique to support our compositional verification. As shown in Fig. 4(1), we extend the existing model-checker SPIN [10] to support model-checking with PPTL_{st}. To do so, an automatic tool PPTL2Never is developed for translating PPTL_{st} formulas to the corresponding Büchi automata which admit all models of the corresponding formulas. The translation is based on an intermediate tableau graph, called Normal Form Graph (NFG for short), which bridges the gap between formulas and the resulting automata, and the translation algorithm can be found in our earlier work [7]. Besides, translator PPTL2Never is integrated within SPIN. In this way, the partial-order model-checking algorithm that has been implemented in SPIN can be used off-the-shelf.

Within the method of partial-order model-checking PPTL_{st}, the system is described in terms of ProMeLa [10] which produces a Büchi automaton BA_s when executed by ProMeLa interpreter within SPIN. The property is first expressed by a PPTL_{st} formula P, then, with the translator, its negation ¬P is transformed into Büchi automaton BA_p, further to Never Claim [10] in the syntax of ProMeLa. In this manner, whether or not the system satisfies property P can be checked by computing the product automaton of BA_s and BA_p, and then checking whether or not the product automaton accepts the empty word. If the word accepted by the product automaton is empty, the system satisfies the property, otherwise the counterexamples are given.

To illustrate how our method works, a simulation program indicating an automatic gas station (AGS) is taken as an example. This program is formed by a
parallel composition of a *pump* process, two *customer* tasks, two *operator* processes and an *pos* subsystem which has been described in the introduction. The architecture of AGS is depicted in Fig. 2(2). And the ProMeLa code of these newly introduced process can be found in Fig. 6. The business process can be described as follows: customers are allowed to prepay (*S*\_swipe) at the pos terminal which connects (*S*\_access) to the remote bank. After the remote bank system finishes money account transfer, the acknowledgement (*S*\_reply) is transmitted to the pos terminal, and is delivered (*S*\_prepay) to an operator process automatically assigned to each customer. The operators compete to access (*S*\_activate) the pump. Once accessed, the pump starts to work (*S*\_refuel). Another customer’s request is held when the pump is refueling. After refueling, the operator charges according to the fuel injection quantity (*S*\_ack) provided by the pump, gives the change (*S*\_return) to the customer, and releases the accession of the pump.

On the business process, it should be guaranteed that operator gives the right change to the right customer after the prepayment. This property can be expressed by PPTL formula $\psi = \diamond ((e_s \text{ prj } p_1) : r_1) \land \diamond ((e_s \text{ prj } p_2) : r_2)$, where $p_i$ is defined as $usr = cus_i$, $r_i$ as $rightchange_i = true$. The sub-formula $(e_s \text{ prj } p_i)$ can indicate that a prepayment is made by customer $cus_i$, since it is proved that in the pos subsystem, the prepayment has been done when the customer swipes card (see the introduction). So the fact that operator gives right change to $cus_i$ can be expressed by formula $(e_s \text{ prj } p_i) : r_1$. Meanwhile, though each customer makes a prepayment, this prepayment may not occur at the beginning in executions. Therefore we employ ‘sometimes’ operator $\diamond$ to precisely capture this property.

With our method, the negation of the property

\[
\neg(\diamond((e_s \text{ prj } p_1) : r_1) \land \diamond((e_s \text{ prj } p_2) : r_2))
\]

...
\(-\psi\) is first transformed to a Büchi automaton (see Fig.5(2), where the centric circle denotes accepting location), and then the Büchi automaton is expressed in Never Claim in Fig.5(1). Given the ProMeLa program of the protocol and the Never Claim, by partial-order reduction, SPIN declares the property violated and outputs a run that contains the violation. The run is visualized as a message sequence of chart, as shown in Fig.4(2). The output shows that 41 states and 65 transitions are traversed. We will explain the reason for the violation in next section.

4.2 Compositional Verifying AGS with PPTL$_{st}$

To illustrate compositional verification with projection construct, we consider the automatic gas station again. By using partial-order model-checking as presented above the pos subsystem can be prove to satisfy its local property, i.e. property defined by formula \(\varphi\) given in the introduction. Then, according to the local property, the subsystem is abstracted. And in the architecture of AGS the pos system can be replace by the abstracted one, as shown in Fig. 6, so that the irrelevant detail is avoided and thus the state space is reduced when verifying the global system, meanwhile the replacement does not affect the overall verification.

![Figure 6: Simulation program and architecture of abstracted automatic gas station](image)

Figure 6: Simulation program and architecture of abstracted automatic gas station

Also by using partial-order model-checking, as shown in Fig.7, it proves that the abstracted program violates the overall property, i.e. the property defined by formula \(\psi\) presented in the above subsection, and an execution as the witness of the violation is obtained. This is because the value of proposition \(q_i\) \((i = 1, 2)\) is always kept false when the operator gives a customer’s change to another (Recall \(q_i\) is defined as \(payer = cus_i\) in the introduction). The violation can be corrected by replacing the operator process in Fig.6 by that in Fig.7(2). In the upgraded process, actions of acknowledgement \(S_{ack}\) and giving change \(S_{return}\) are bound together to form an atomic sequence. Therefore, in all executions of the gas station program, actions of swiping card \(S_{swipe}\), refueling \(S_{refuel}\) and giving change for the same customer are sequentially bridged.

![Figure 7: Violation and revised operator process](image)

Figure 7: Violation and revised operator process

Note that the gas station program is verified in a compositional manner because the pos subsystem is first verified against a local property which relates to the overall property of the global system, and the pos subsystem is abstracted according to the local property, further used in verification of the global system (see the introduction). With this verification procedure, projection construct plays a significant role.

5 Conclusion

In this paper, we first proposed stutter-invariant propositional projection temporal logic PPTL$_{st}$, and proved it can express all the stutter-invariant properties expressible in PPTL. The method of determining stutter-invariance of PPTL formulas is also given. Then, as discussed in the introduction the projection construct which characterizes PPTL$_{st}$, is mainly used to support compositional verification. We illustrated this feature by an example—automatic gas station.

Our experience has shown that PPTL$_{st}$ is useful in compositional verification when formulas specifying each module and the global system are provided. However, in practice properties of the overall system are usually provided other than these for modules. Hence, there is a tedious job left for the designer to imply modules’s specifications from the overall ones. This burden is undesirable. Thus, our future research is to study the method of extracting modular specification from the overall specification based on the system’s structure. Further, we will apply PPTL$_{st}$ in compositional verification for large-scale distributed programs, such as Russian Cards Protocol [26, 27] we have just developed.

References: