Formal verification with projection temporal logic

TIAN Cong & DUAN ZhenHua*
ICTT and ISN Laboratory, Xidian University, Xi'an 710071, China
Received November 18, 2014; accepted December 2, 2014

Abstract Projection temporal logic (PTL) is an extension of interval temporal logic (ITL) with a new projection operator \( \text{prj} \) and infinite intervals which has been well investigated in the past ten years. In this paper, we review the work on PTL in four aspects: (1) decidability, complexity and expressiveness of propositional PTL (PPTL); (2) modeling, simulation and verification language (MSVL); (3) formal verification approaches with MSVL and PPTL; and (4) supporting toolkit MSV.

Keywords Projection temporal logic; Modeling; Verification; Semantics; Model checking

1 Introduction

With the development of computer technology, both hardware and software grow increasingly in size and functionality. This makes it hard to find bugs residing in systems by means of the traditional testing and simulation approaches. Formal verification is a rigorous technique based on sound mathematics. There are mainly two formal verification approaches, model checking \([1,2]\) and theorem proving \([3]\).

Model checking is an automatic verification approach which can automatically determine whether or not a model of a system logically satisfies a specification through exploring the model in an exhaustive way. Two of the most famous model checkers are SPIN \([4]\) and SMV (NuSMV) \([5]\). Due to the fully automatic verification mode, Model checkers are easy to be adopted by industry. However, it suffers from the state explosion problem. To combat this problem, various approaches have been proposed such as symbolic \([6]\), bounded \([7]\), partial-order \([8]\) and abstract model checking \([9]\). Theorem proving is built on proof theory. It checks whether or not a system satisfies a desired property by means of proving whether or not a theorem is valid in an axiomatic system. With theorem proving approach, a sound (often also complete) proof system of some mathematical logic is established first. Then both the system to be verified and the desired property are specified as formulas in this logic. Theorem proving can be applied to both finite and infinite systems. Details of the proof enable users to learn more about the systems. Nevertheless, it often requires expertise of human to assist the verification process. Famous theorem provers include PVS \([10]\), Coq \([11]\), and ACL2 \([12]\).

In both model checking and theorem proving approaches, temporal logics \([13]\) play a significant role. Classical temporal logics are linear temporal logic (LTL) \([14]\), computation temporal logic (CTL) \([1]\), temporal logic of actions (TLA) \([15]\), interval temporal logic (ITL) \([16]\), and so on. Projection temporal logic (PTL) \([17]\) is an extension of ITL with a new projection operator \( \text{prj} \) \([18]\) and infinite intervals where \( \text{prj} \) can subsume chop operator. In \( (P_1, \ldots, P_n) \text{ prj } Q \), all the formulas \( P_1, \ldots, P_n \) and \( Q \) are autonomous and can specify their own interpreted interval. \( Q \) is interpreted in a parallel fashion with \( P_1, \ldots, P_n \), while \( P_i \)'s are interpreted in a sequential manner. In this way, projection construct can describe two different time scales which is useful in expressing concurrent computations with multi-tasking and multi-processors. As a propositional subset of PTL, propositional PTL (PPTL) is useful in specifying properties of systems to be verified.

* Corresponding author. Email: zhbduan@mail.xidian.edu.cn, ctian@mail.xidian.edu.cn

DOI: 10.16262/j.cnki.1005-0841.2014.02.002
• Reviews •

The demand for modeling, specification and verification [19] has led to the emergence of temporal logic programming languages, for instance XYZ/E [20], METATEM [21], Tempura [16], MSVL (modeling, simulation, and verification language) [22,23], and so on. Among them, MSVL is an executable subset of PTL. In MSVL, the framing operator frame is proposed for memory management. Various data types, function calls, as well as synchronous and asynchronous communication mechanisms make MSVL more practical. As a result, all constructs in C language are involved in MSVL. This enables us to verify C programs via MSVL. Besides, Verilog/VHDL languages can also be translated into MSVL, which reduces the verification of Verilog/VHDL programs into the verification of MSVL programs.

In this paper, we review the work on PTL in four aspects: (1) decidability, complexity and expressiveness of PPTL; (2) MSVL language as well as minimal model, operational and axiomatic semantics of MSVL; (3) formal verification approaches with MSVL and PPTL; and (4) supporting toolkit MSV for verification with MSVL.

The rest of the paper is organized as follows. Section 2 briefly introduces the underlying logic PTL. In Section 3, decidability, complexity and expressiveness results about PPTL are discussed. Next, in Section 4, MSVL as well as its semantics are presented. Then, model checking and theorem proving techniques with PPTL and MSVL are presented in Section 5, and the supporting toolkit is illustrated in Section 6. Finally, conclusions are drawn in Section 7.

2 Projection temporal logic

This section presents syntax and semantics of projection temporal logic.

2.1 Syntax

Let Prop be a countable set of atomic propositions and V a countable set of variables. B = {true, false} represents the boolean domain while D denotes all the data we need. The term e and formula P of PTL are presented by the following grammar:

\[ e ::= c \mid \nu \mid \bigcirc e \mid \bigcap e \mid f(e_1, \ldots, e_n) \]

\[ P ::= = \mid e_1 = e_2 \mid P(e_1, \ldots, e_n) \mid \neg P \mid P_1 \land P_2 \]

\[ \exists \nu : P \mid \bigcirc P \mid (P_1, \ldots, P_n)_{\text{prj}} Q \mid P^+ \]

where \( c \in D \) is a constant, \( p \in \text{Prop} \) a proposition, and \( \nu \in V \) a static or dynamic variable. In \( f(e_1, \ldots, e_n) \) and \( F(e_1, \ldots, e_n) \), \( f \) stands for a function and \( F \) indicates a predicate. \( P, P_1, \ldots, P_n, \) and \( Q \) are well-formed PTL formulas. Operators \( \rightarrow, \land \) and \( \exists \) are similar to the ones in classical first order logic, while \( \bigcirc(\text{next}) \), \( \bigcap(\text{previous}) \), + (chop-plus) and prj (projection) are temporal operators. A formula (term) is called a state formula (term) if it does not contain any temporal operators; otherwise it is a temporal formula (term). A static variable remains the same over an interval whereas a dynamic variable can have different values at different states.

2.2 Semantics

A state \( s \) is a pair of assignments \( (I_{\text{var}}, I_{\text{prop}}) \), where \( s[\nu] = I_{\text{var}}[\nu] \) for each variable \( \nu \in V \) and \( s[p] = \bigcap_{\text{prj}} p \) for each proposition \( p \in \text{Prop} \). Here \( I_{\text{var}}[\nu] \) assigns \( \nu \) a value within the data domain \( D = D \cup \text{nil} \) with appropriate type, where \( \text{nil} \) means undefined, and \( I_{\text{prop}}[p] \) sets \( p \) a value in \( B \). 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 \( \omega \). To have a uniform notation for both finite and infinite intervals, we will use extended integers as indices, that is, \( N_\omega = N_\omega \cup \{\omega\} \) and extend the comparison operators, \( =, <, \leq, \) to \( N_\omega \) by considering \( \omega = \omega \) and for all \( i \in N_\omega, i < \omega \). Moreover, we write \( \exists \sigma \leq \omega \) as \( \exists \sigma < \omega \).

To define the semantics of the projection construct, we need an auxiliary operator \( \downarrow \). Let \( \sigma = (s_0, s_1, \ldots) \) be an interval and \( r_1, \ldots, r_k \) integers \( (k \geq 1) \) such that \( 0 \leq r_i, \ldots, r_k \leq \omega \). The projection of \( \sigma \) onto \( r_1, \ldots, r_k \), \( r \) is the projected interval, \( r \downarrow (r_1, \ldots, r_k) \downarrow (s_1, s_2, \ldots, s_\omega) \), where \( s_i, \ldots, s_i \) are attained from \( r_i, \ldots, r_k \) by deleting all duplicates. In other words, \( t_1, \ldots, t_k \) is the longest strictly increasing subsequence of \( r_1, \ldots, r_k \).
• Reviews

For instance, \( \langle s_0, s_1, s_2, s_3 \rangle \downarrow (0, 2, 2, 3) = \langle s_0, s_2, s_3 \rangle \). The concatenation (\( \cdot \)) of an interval \( \sigma = \langle s_0, s_1, \ldots, s_\ell \rangle \) with another interval \( \sigma' = \langle s_0', s_1', \ldots, s_\ell' \rangle \) is represented by \( \sigma \cdot \sigma' = \langle s_0, s_1, \ldots, s_\ell, s_0', s_1', \ldots, s_\ell' \rangle \) (not sharing any states). To evaluate the existential quantification, an equivalence relation is required.

For a variable \( v \), we can write \( \sigma' \models_\sigma v \) if \( \sigma' \) is an interval that is the same as \( \sigma \) except that different values can be assigned to \( v \).

An interpretation for a PTL term or formula is a tuple \( \mathcal{I} = (\sigma, i, k, j) \), where \( \sigma = \langle s_0, s_1, \ldots \rangle \) is an interval, \( i \) and \( k \) are non-negative integers, and \( j \) is an integer or \( \omega \) such that \( i \leq k \leq j \leq |\sigma| \). We write \( (\sigma, i, k, j) \) to mean that a term or formula is interpreted over a subinterval \( \sigma_{i, \ldots, j} \) with the current state being \( s_k \). We use \( \mathcal{I}_n \) and \( \mathcal{I}_p \) to denote the state interpretation at state \( s_k \). Each \( m \)-place function symbol \( f \) has an interpretation \( \mathcal{I}[f] \) which is a function mapping \( m \) elements in \( D^m \) to a single value in \( D' \). Interpretations of predicate symbols \( \mathcal{I}[F] \) are similar but map to truth values. We assume that \( \mathcal{I} \) interprets operators such as \( +, -, \cdot, /, \land, \lor, \leq, \geq, = \), etc. standardly. The evaluation of \( \varepsilon \) relative to \( \mathcal{I} = (\sigma, i, k, j) \) is defined as \( \mathcal{I}[\varepsilon] \) as shown in Table 1 while the satisfaction relation \( \models \) of formulas is given in Table 2.

A formula \( P \) is satisfied by an interval \( \sigma \), denoted by \( \sigma \models P \) if \( \sigma, 0, 0, |\sigma| \models P \). A formula \( P \) is called satisfiable if \( \sigma \models P \) for some \( \sigma \). Furthermore, \( P \) is said to be valid, denoted by \( \models P \), if \( \sigma \models P \) for all intervals \( \sigma \).

### Table 1 Interpretation of PTL terms

<table>
<thead>
<tr>
<th>( \mathcal{I} [c] )</th>
<th>( c \in D )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{I}[e] = s_k[v] = \mathcal{I}_n[I] )</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{I}[f(e_1, \ldots, e_m)] = \begin{cases} \mathcal{I}_n[I(e_1), \ldots, I(e_m)] &amp; \text{if } \mathcal{I}_n[I(e_1), \ldots, I(e_m)] \neq \text{nil for all } 1 \leq h \leq m \ \text{nil} &amp; \text{otherwise} \end{cases} )</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{I}[\varepsilon] = \begin{cases} (\sigma, i, k+1, j)[\varepsilon] &amp; \text{if } k &lt; j \ \text{nil} &amp; \text{otherwise} \end{cases} )</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{I}[\sigma] = \begin{cases} (\sigma, i, k-1, j)[\sigma] &amp; \text{if } k &lt; j \ \text{nil} &amp; \text{otherwise} \end{cases} )</td>
<td></td>
</tr>
</tbody>
</table>

### Table 2 Interpretation of PTL formulas

| \( \mathcal{I} \models P \) iff \( \mathcal{I}_n[I] = \text{true} \) |
| \( \mathcal{I} \models F(e_1, \ldots, e_m) \) iff \( \mathcal{I}[F(I[e_1], \ldots, I[e_m])] = \text{true} \), and \( \mathcal{I}_n[I] \neq \text{nil for all } 1 \leq h \leq n \) |
| \( \mathcal{I} \models e_1 = e_2 \) if \( \mathcal{I}[e_1] = \mathcal{I}[e_2] \) |
| \( \mathcal{I} \models \neg P \) iff \( \mathcal{I} \models \neg P \) |
| \( \mathcal{I} \models P \land Q \) iff \( \mathcal{I} \models P \) and \( \mathcal{I} \models Q \) |
| \( \mathcal{I} \models \exists v P \) iff \( \mathcal{I} \models P \) for some \( \sigma \models v \) |
| \( \mathcal{I} \models (P_1, \ldots, P_m) \) per Q |
| \( \mathcal{I} \models Q \) iff there are integers \( r_0, \ldots, r_m \) and \( k = r_0 \leq \cdots \leq r_m \leq j \) such that \( (\sigma, i, r_0, r_1) \models P_1 \), \( (\sigma, r_0, 0, r_1) \models P_2 \) for all \( 1 \leq i \leq m \), and \( \mathcal{I}[\sigma, \sigma'] = Q \) for \( \sigma' \) given by:
| (1) \( r_0 < j \) and \( \sigma' = \sigma \upharpoonright (r_0, \ldots, r_m) \cdot \sigma(r_0+1, \ldots, j) \) |
| (2) \( r_0 < j \) and \( \sigma' = \sigma \upharpoonright (r_0, \ldots, r_m) \) for some \( 1 \leq h \leq m \) |
| \( \mathcal{I} \models P' \) iff there are finitely many integers \( r_0, \ldots, r_m \) and \( k = r_0 \leq \cdots \leq r_m \) such that \( \mathcal{I}[\sigma, r_0, 0, r_1, \ldots, r_m] \models P \) for all \( 1 \leq i \leq n \) or \( j = \omega \) and there are infinitely many integers \( k = r_0 \leq r_1 \leq \cdots \leq r_m \) such that \( \lim r_i = \omega \) and for all \( j \geq 1 \), \( (\sigma, r_0, 0, r_1, \ldots, r_m) \models P \). |

### 2.3 Derived formulas

Some derived operators are formalized in Table 3 for the sake of convenience. The abbreviations true, false, \( \lor \), \( \rightarrow \), and \( \leftrightarrow \) are defined as usual while others are explained in [22, 17]. Sometimes, \( \models \Box (P \leftrightarrow Q) \) is represented by \( P \equiv Q (\text{strong equivalent}) \), meaning that \( P \) and \( Q \) have the same truth-value at all states in every model.
3 Propositional projection temporal logic

This section presents PPTL as well as its decidability, complexity and expressiveness results.

3.1 Syntax and semantics

Propositional projection temporal logic (PPTL) \([22,24-27]\) is a propositional subset of PTL with \(\bigcirc\), \(\text{prj}\) and \(+\) as its basic temporal operators. It is an extension of propositional interval temporal logic (PITL). The syntax of PPTL is defined below:

\[
P ::= p \mid \neg p \mid P_1 \land P_2 \mid \bigcirc P \mid (P_1, \ldots, P_n)\text{prj} Q \mid P^*
\]

Semantics of PPTL formulas is similar to the one of PTL. Some derived formulas, such as \(\epsilon\), \(\text{more}\), \(\text{fin}\), \(\text{halt}\), \(\text{keep}\), and \(P^*\), can also be obtained in PPTL. In order to deal with PPTL formulas in a unified way, normal form of PPTL is formalized as follows:

Definition 1 (Normal Form, NF): Let \(Q_n\) be the set of atomic propositions appearing in a PPTL formula \(Q\). Normal form of \(Q\) can be defined by:

\[
Q = \bigvee_{i=1}^n (Q_{a_i} \land \epsilon) \lor \bigvee_{i=1}^n (Q_{a_i} \land \bigcirc Q_i')
\]

where \(Q_{a_i} = \bigwedge_{j=1}^n q_{a_j} \land Q_{a_i} = \bigwedge_{j=1}^m q_{a_j} \land q_{a_j} \in Q_a\) for any \(a \in Q\), \(\epsilon\) denotes \(r\) or \(\neg r\); each \(Q_i'\) is a general PPTL formulas without \(\bigvee\) being the main operator.

For convenience, we call \(\bigvee_{j=1}^n (Q_{a_j} \land \epsilon)\) terminal products and \(\bigvee_{i=1}^n (Q_{a_i} \land \bigcirc Q_i')\) future products. In addition, \(Q_{a_i}\) and the \(Q_i'\) are present components and each \(\bigcirc Q_i'\) is a future component \((1 \leq i \leq r)\).

Theorem 1: Any PPTL formula \(P\) can be transformed to its normal form.

Based on Theorem 1, we can obtain an algorithm NF for transforming a PPTL formula into its normal form, whose details can refer to \([24,26]\). The normal form graph (NFG) of a PPTL formula is inductively defined below.

Definition 2: (Normal Form Graph, NFG) For a PPTL formula \(P\), the set \(CL(P)\) of nodes and the set \(EL(P)\) of edges connecting nodes in \(CL(P)\) are inductively defined as follows:

1) Initially, let \(V_0 = CL(P) = EL(P) = \emptyset\).

2) Let \(P = \bigvee_i P_i\). For each \(i\), \(P_i \in V_0\), \(P_i \in CL(P)\).

3) For all \(Q \in CL(P) \setminus \{\epsilon, \text{false}\}\), if \(Q\) is rewritten into its normal form \(\bigvee_{j=0}^n (Q_{a_j} \land \epsilon) \lor \bigvee_{i=0}^n (Q_{a_i} \land \bigcirc Q_i')\), then \(\epsilon \in CL(P)\), \((Q, Q_{a_i}, \epsilon) \in EL(P)\) for each \(j\), \(0 \leq j \leq n\); \(Q_i' \in CL(P)\), \((Q, Q_{a_i}, Q_i') \in EL(P)\) for all \(i\), \(0 \leq i \leq k\).

The NFG of formula \(P\) is the directed graph \(G = (CL(P), EL(P), V_0)\).

In an NFG, any root node in \(V_0\) is denoted by a circle without any input edge, \(\epsilon\) node is marked by a small black dot, and each of other nodes by a single circle. Each edge is denoted by a directed arc connecting two nodes. An example of NFG is shown in Figure 1. A finite path is a finite alternating sequence of nodes and edges, \(\pi = \langle n_0, e_0, n_1, e_1, \ldots, e_r \rangle\) from a root node to the \(\epsilon\) node, while an infinite path is an infinite alternating sequence of nodes and edges, \(\pi = \langle n_0, e_0, n_1, e_1, \ldots, n_i, e_i, \ldots, n_j, e_j, \ldots, n_z, e_z, \ldots \rangle\) departing from the root node with some nodes, e.g. \(n_i, \ldots, n_j\), occurring for infinitely many times. For convenience, we use \(\text{Inf}(\pi)\) to denote the set of nodes which infinitely often occur in the infinite path \(\pi\). Under
some circumstances, in a path of NFG of formula $Q$, a node $n_i$ can be replaced by a formula $Q_i \in CL(Q)$ and an edge $e_i$ can be replaced by a state formula $Q_e \in EL(Q)$. For a given path $\pi$, we can define the $i^{th}$ prefix $\pi^{'i}$ and the $i^{th}$ suffix $\pi^{'(i)}$ of $\pi$.

Figure 1 An illustrative example of NFG.

3.2 Decidability of PPTL

Satisfiability and validity of formulas are fundamental issues in the model theory of a logic. Particularly, satisfiability plays an important role in the model checking approach since model checking usually proceeds by checking the satisfiability of the negation of a property described in a temporal logic. For the decidability problem of interval based temporal logic, many researchers devoted themselves into this field. In 1983, Halpern and Moszkowski demonstrated that the satisfiability for PITL is undecidable [28]. In 2003, Bowman and Thompson proved that the satisfiability of PPTL over finite intervals is decidable [29] and gave a tableau-based decision procedure. In 2008, we proved that the satisfiability of PPTL over both finite and infinite intervals is decidable [24], which makes model checking with PPTL possible. Meanwhile, we present a decision procedure for checking the satisfiability of PPTL formulas.

Actually, finite paths in the NFG of PPTL formula $Q$ precisely characterize finite models of $Q$. For infinite models, we need to ensure the finiteness of $P$ in strong chop construct $P_1 Q$, called FSC_Property. That is, it is required that $P_1 Q$ can be reduced to some formula $P_1 \land \epsilon_1 Q$ by means of repeatedly using normal form reduction and $P_1 \land Q$ is satisfiable. To capture this, we formalize the FSC_Property of $P_1 Q$ by means of two functions FSC and fsc. They are defined as FSC (also $fsc$) : $\Sigma \times \Pi_{\text{path}} \rightarrow \{true, false\}$, where $\Pi_{\text{path}}$ represents the set of all path formulas while $\Sigma$ denotes all paths in NFGs of all formulas of $\Pi_{\text{path}}$. $fsc(\pi, R) = true$ if (1) $R = R_{\emptyset}$; (2) $R = P_1 Q$ and there exists $i \in N_0$ such that $R_{\emptyset} = P_{\emptyset} \land \epsilon_1 Q$; (3) $R = P \land Q$, $fsc(\pi, Q) = true$ and $fsc(\pi, P) = true$; or (4) $R = P \lor Q$ and $fsc(\pi, P) = true$ (or $fsc(\pi, Q) = true$). Accordingly, $FSC(\pi, R) = true$ if $fsc(\pi^{(k)}, R_k) = true$ for all $k \in N_0$.

Then we obtain the following conclusion that plays a significant role in the decidability of PPTL formulas.

Theorem 2: In the NFG of PPTL formula $Q$, finite paths precisely characterize finite models of $Q_i$ infinite path $\pi$ with $FSC(\pi, Q) = true$ precisely characterizes infinite models of $Q$.

To explicitly display whether or not the FSC_Property of a chop formula is satisfied, extra propositions $l_k$, $k \in N_0$ and $k > 0$, are introduced. Let $Prop_0 = \{l_1, l_2, \ldots\}$ be the set of extra propositions. Note that these extra propositions are merely employed to mark nodes and not allowed to appear in a PPTL formula. When constructing NFGs from normal form reductions, for any chop formula $P_1 Q$, we equivalently rewrite it as $P \land fin(l_k) \land Q$. As a result, by using $fin(l_k)$, FSC_Property of $P_1 Q$ is satisfied if there exists an edge where $l_k$ holds. Furthermore, $fin(l_k)$ occurring in a node $P \land fin(l_k) \land Q$ means that FSC_Property of $P_1 Q$ has not been satisfied at this node. Accordingly, labeled normal form graph (LNFG) is defined based on NFG by means of $l_k$ propositions.

Definition 3: (Labeled Normal Form Graph, LNFG) For a PPTL formula $P$, its LNFG is a tuple $G = (CL(P), EL(P), V_0, L = \{L_1, \ldots, L_m\})$, where $CL(P)$, $EL(P)$ and $V_0$ are identical to the ones in the NFG, while each $L_k \subseteq CL(P)$, $1 \leq k \leq m$, is the set of nodes with $l_k$ labels. □

An illustrative example of LNFG is showed in Figure 2. Algorithm LNFG based on Algorithm NFG
[24] is formalized by further rewriting any chop component $P_1Q$ as $P \land fin(l_1)1Q$ for some $k \in N_0$ whenever it is encountered. The details can be found in [26].

\[
\begin{align*}
    n_0 & : q \land (\text{empty} \land fin(l_1); q) \land (p \land \text{empty}; q) \\
    n_1 & : q \land (\text{empty} \land fin(l_2); q) \land (p \land \text{empty}; q)
\end{align*}
\]

**Figure 2** An illustrative example of LNFG.

Theorem 3: In the LNFG $G$ of a formula $P$, finite paths precisely characterize finite models of $P$; infinite paths with $\text{Inf}(\pi) \subseteq L_1$, for all $1 \leq i \leq m$ precisely characterize infinite models of $P$.

Consequently, a decision procedure for checking the satisfiability of a PPTL formula $P$ can be given based on the LNFG of $P$. Firstly, the LNFG of a PPTL formula $P$ is constructed; if there exists $\varepsilon$ node in $CL(P)$, $P$ is satisfiable with finite models; if there exists infinite path $\pi$ with $\text{Inf}(\pi) \subseteq L_1$, for all $1 \leq i \leq m$, $P$ is satisfiable with infinite models; otherwise, $P$ is unsatisfiable. With this method, the decidability problem of PPTL over both finite and infinite intervals is well handled.

Since PPTL can subsume PTL, the decision procedure for PPTL can also be used to check the satisfiability of PTL formulas with minor changes. In this way, the decidability problem of PTL over both finite and infinite intervals is well handled. Subsequently, to achieve great efficiency, the decision procedure for PPTL formulas has been improved [26]. The decision procedure works well and a practical model checker based on SPIN has been developed.

### 3.3 Complexity of PPTL

In [30], Stockmeyer shows that the emptiness problem for star-free expressions is non-elementary. He also claims that if this problem is efficiently reducible to a particular decision problem, the decision problem is non-elementary. Thus, through reducing the emptiness problem of star-free expressions to the problem of the satisfiability of PPTL and PTL formulas, we demonstrate that the lower bound for the complexity of the satisfiability of PPTL and PTL formulas is non-elementary [27]. To the best of our knowledge, this is the first systematic proof of the complexity of interval based propositional temporal logics.

### 3.4 Expressiveness of PPTL

We investigate the expressiveness of PPTL and PTL and the characterizations of its fragments in [25]. To this end, Büchi Automata and Regular Expression are firstly extended to Stutter Büchi Automata (SBA) and extended regular expressions (ERE). Then PPTL is proved to exactly express the full regular language by three transforming procedures among PPTL, SBA, and ERE. In other words, each language defined by a PPTL formula can be defined by an SBA; each language defined by an SBA can be defined by an ERA; each language defined by an ERE can be defined by a PPTL formula. As a result, the expressiveness of PPTL is equivalent to that of the full regular language.

Subsequently, a hierarchy of expressiveness for various PPTL operators is described, which enables us to measure and improve model checking algorithms for PPTL. In fact, PPTL and its fragments can be classified into the following different language classes. Let an expression like $L(\Delta, T)$ refer to the specific fragment of PPTL with temporal operators $\Delta, T$ and the basic connections in typical propositional logic. Then we have: (1) $L(\varepsilon)$ has the same expressiveness as star-free regular expressions without $\varepsilon$; (2) $L(\text{O}, 1)$ and $L(\text{O}, \text{prj})$ have the same expressiveness as star-free regular expressions; (3) $L(\varepsilon, 1, *)$ has the same expressiveness as regular expressions without $\varepsilon$; (4) $L(\text{O}, 1, 1, *)$, $L(\text{O}, 1, \text{prj})$, and $L(\text{O}, 1, 1, *, \text{prj})$ has the same expressiveness as full regular expressions. In addition, the expressiveness of PTL is also investigated. Particularly, $L(\text{O}, 1, \text{proj})$ has the same expressiveness as $L(\text{O}, \text{prj})$, which also reveals that proj in PPTL can subsume; and proj in PTL from the expressiveness standpoint.

42 Vol. 22, No. 2, 2014 SCIENCE FOUNDATION IN CHINA
• Reviews •

3.5 Discussion

Compared with LTL [14] and CTL [1] PPTL can express full regular language [25] while PLTL and PCTL fail to do so. Thus, PPTL can specify the following two kinds of properties that cannot be or can be hardly expressed in LTL and CTL; (1) interval-sensitive properties, such as ‘p holds between the 15th and 20th time units’, which can be formalized as \( \text{len}(15) \land p \land \text{len}(20) \); (2) star properties, such as ‘p holds at the even states regardless of odd states’, which can be specified as \((p \land \bigcirc^k (p \land \text{empty}))^*\).

Further, compared to PITL, the projection operator in PPTL is more powerful than the original projection operator in PITL. Moreover, both PPTL and mu-calculus [31] have the expressiveness of full regular language. However, the fixed-points in mu-calculus is more intricate, which makes mu-calculus not convenient to be used.

4 Programming language MSVL

MSVL [22, 32] is a modeling, simulation and verification Language, which provides an executable PTL framework with more succinct description and immediate practical application. It extends Tempura [16] with a new projection \( \text{pj} \) and infinite models.

4.1 Statements in MSVL

In MSVL, expressions can be regarded as PTL terms while statements can be considered as PTL formulas. The arithmetic expression \( e \) and boolean expression \( b \) of MSVL are inductively defined as follows, where \( c \) is a constant, \( x \) a variable:

\[
\begin{align*}
  e & ::= c \mid x \mid \bigcirc x \mid \bigotimes x \mid e_0 \ op \ e_1 \quad (op ::= -+|\neg|\ast) \\
  b & ::= \text{true} \mid \text{false} \mid \neg b \mid b_0 \land b_1 \mid e_0 = e_1 \mid e_0 < e_1
\end{align*}
\]

A program in MSVL can be formalized in Table 4. Traditionally, \( x \) denotes a variable, \( e \) stands for an arbitrary arithmetic expression, \( b \) represents a boolean expression, and \( P_1, \ldots, P_n, P, \) and \( Q \) are general programs.

<table>
<thead>
<tr>
<th>Table 4 MSVL programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
</tr>
<tr>
<td>Assignment</td>
</tr>
<tr>
<td>Positive Immediate Assignment</td>
</tr>
<tr>
<td>Projection</td>
</tr>
<tr>
<td>Sequential statement</td>
</tr>
<tr>
<td>State Frame</td>
</tr>
<tr>
<td>Interval Frame</td>
</tr>
<tr>
<td>Conjunction</td>
</tr>
<tr>
<td>Selection</td>
</tr>
<tr>
<td>Next statement</td>
</tr>
<tr>
<td>Always statement</td>
</tr>
<tr>
<td>Conditional statement</td>
</tr>
<tr>
<td>Existential Quantification</td>
</tr>
<tr>
<td>While statement</td>
</tr>
<tr>
<td>Parallel</td>
</tr>
<tr>
<td>Synchronized Communication</td>
</tr>
</tbody>
</table>

‘empty’ is the termination statement, which simply states that the current state is the final state of the interval over which a program is executed. The next statement ‘next P’ means that \( P \) holds at the immediate successive state. ‘always P’ implies that \( P \) is always true in all states from now on. The sequential statement ‘\( P_1 \cdot Q \)’ signifies a computation of \( P \) followed by \( Q \), that is, \( P \) keeps executing from the current state until some point in future at which it terminates and \( Q \) will start executing. The conditional statement ‘if \( b \) then \( P \) else \( Q \)’ and while statement ‘while \( b \) do \( P \)’ can be illustrated as that in the conven-
tional imperative languages. 'if b then P else Q' first evaluates the boolean expression; if b is true, the process P is executed; otherwise Q is executed. The iteration 'while b do P ' allows process P to be repeatedly executed a finite (or an infinite) number of times over a finite (resp. an infinite) interval as long as the condition b is satisfied at the beginning of each execution. If b becomes false, the whilestatement terminates. In the assignment statement 'x := e', if e is evaluated to a constant in D and x has not been specified at the current state or was specified to the same value as e, we say x is unified with e. 'P or Q' represents the selection statement asserting that P or Q is executed with non-determinacy. The existential quantification statement 'exist x : P(x)' intends to hide the variable x within the process P and may allow a process P to take advantage of a local variable.

The conjunction statement 'P and Q' declares that the processes P and Q are executed concurrently sharing all the states during the mutual execution. The parallel construction 'P || Q' shows another concurrent computation manner. The distinguished difference between 'P || Q' and 'P and Q' is that the former permits both P and Q to be autonomous, or rather to specify their own intervals while the latter does not. For example, len(2) || len(3) holds but len(2) , len(3) is obviously false. Projection can be treated as a special parallel computation with greater autonomy, which is executed on two different time scales. '(P_1, ..., P_n)proj Q' tells us that Q is executed in parallel with P_1, ..., P_n over an interval obtained by taking the endpoints of the intervals over which the P_i's are executed. In this construct, processes P_1, ..., P_n, Q are self-governing and each of them has the right to specify the interval over which it is executed. In particular, the sequence of P_i's and Q may terminate at different points.

Framing concerns how the value of a variable can be carried from one state to the next. The crux is how to perceive the assignments of values to variables. To identify an occurrence of an assignment to a variable, say x, we make use of a flag called the assignment flag, denoted by predicate af(x); it is true whenever an assignment of a value to x is encountered, and false otherwise. To define af(x), a new assignment positive immediate assignment x <= e def X = e & p_x is needed, where p_x is an atomic proposition connected with the variable x and cannot be used for other purposes. Then the assignment flag is formalized as af(x) def px. As expected, when x <= e is encountered, p_x is set to true, hence af(x) is true; whereas if no assignment to x takes place, px is unspecified. In this case, we will use the minimal model [22] to force it to be false. Note that the definition given above is one way to specify af(x) and there may be some other methods to formulate it. Armed with af(x) we can define state frame lbf(x) and interval frame frame(x).

Intuitively, 'lbf(x)' means that, when a variable is framed at a state, its value remains unchanged if no assignment is encountered at that state while 'frame(x)' states that a variable is framed over an interval if it is framed at each state over the interval.

Introducing the framing operator enables us to define the synchronizing construct await(c), where c is a condition, i.e., a boolean expression and V_c represents all dynamic variables contained in c. The await-statement is employed to synchronize communication between parallel processes with shared variables. It does not change any variables, but waits until the condition c becomes true, at which point it terminates. Similarly as PPTL, normal form and normal form graph for MSVL can also be defined.

Definition 4 (Normal Form, NF): Let Q be a program in MSVL. The normal form of Q is defined as

\[ Q \|e\| \cup \bigcup_{i=1}^{k} (Q_e \land \varepsilon) \bigcup \bigcup_{j=1}^{h} (Q_d \land O Q_j) \]

where k + h ≥ 1 and the following hold;

- Q_e is a general program;
- each Q_e and Q_d is either true or a state formula of the form \( P_1 \land \cdots \land P_m (m \geq 1) \) such that each \( P_i \) (1 ≤ i ≤ m) is either \( x = c \) with \( x \in V, c \in D \), or \( p_d \) denoting \( p_d \) or \( \neg p_d \).

4.2 Asynchronous communication in MSVL

In order to specify and verify distributed systems, asynchronous communication techniques are introduced into MSVL [33]. To do this, firstly process structure for describing behaviors of systems is presented. Then channel structure for passing messages is defined as a first-in-first-out (FIFO) list. For in-
Reviews

stance, \((1, 2)\) is a channel containing two elements and \((\) is a null channel. Further, based on channels, two couples of asynchronous communication commands are formalized in Table 5, where \(c\) is a channel and \(x\) is a variable; \(\text{isfull}(c)\) and \(\text{isempty}(c)\) are predicates to check the status of the channel \(c\) and are respectively true when \(c\) is full and empty; \text{head} and \text{tail} are functions to respectively fetch the first element and the tail list of \(c\).

<table>
<thead>
<tr>
<th>Table 5 Asynchronous communication commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>send((c, x)) (\equiv) \text{await}(\neg \text{isfull}(c)) \land c \leftarrow c \cdot \langle x \rangle</td>
</tr>
<tr>
<td>receive((c, x)) (\equiv) \text{await}(\neg \text{isempty}(c)) \land x = \text{head}(c) \land c = \text{tail}(c)</td>
</tr>
<tr>
<td>put((c, x)) (\equiv) if((\neg \text{isfull}(c))) then ((c \leftarrow c \cdot \langle x \rangle) ) else {skip}</td>
</tr>
<tr>
<td>get((c, x)) (\equiv) if((\neg \text{isempty}(c))) then ((x \leftarrow \text{head}(c)) \land c = \text{tail}(c)) else {skip}</td>
</tr>
</tbody>
</table>

Commands ‘send’ and ‘put’ are for sending messages while commands ‘receive’ and ‘get’ for receiving messages. The command ‘send\((c, x)\)’ inserts the message \(x\) into the tail of channel \(c\) when \(c\) is not full. If \(c\) is full, the statement will wait until \(c\) has an empty place for \(x\), ‘receive\((c, x)\)’ removes the first message from \(c\) and stores it into \(x\) when \(c\) is not empty. If \(c\) is empty, it will wait until some messages appear in \(c\). Further, ‘put\((c, x)\)’ firstly determines whether or not the channel \(c\) is full; if this is true, it terminates without doing anything; otherwise, it puts \(x\) at the end of \(c\). Moreover, at first, ‘get\((c, x)\)’ decides whether or not the channel \(c\) is empty; if it is empty, it does nothing; or else, it acquires the head message from \(c\) and stores it into \(x\).

4.3 Semantics of MSVL

Formal semantics of a temporal logic programming language is an essential prerequisite for formal verification and facilitates analyzing programs in a rigorous way. Semantics of a program in imperative languages can be captured in an operational or denotational or axiomatic manner. In temporal logic programming, these semantics of a program can also be investigated. Since a temporal logic programming language is a subset of the corresponding logic and the logic has its model theory and axiomatic system, the semantics of a program can be captured naturally by the model theory and axiomatic theory respectively. Of course, when executed, a program can also be interpreted in a more operational way. Hence, we will introduce the minimal model semantics [34], operational semantics [35] and axiomatic semantics [36] of MSVL. The consistencies between minimal model semantics, operational semantics and axiomatic semantics have been proved.

1) Minimal Model Semantics of MSVL: In [34], the minimal model semantics of MSVL is investigated. Canonical models are used to define the semantics of non-framed programs. In general, a canonical interpretation on propositions is a subset \(I_{prop}\) that does not include those false propositions. Let \(\sigma = \langle (I_{ar}, I_{prog}, I_{prop}), (I_{ar}, I_{prop})^{*}\rangle\) be a model. We denote the sequence of interpretation on propositions of \(\sigma\) by \(\sigma_{prop} = \langle I_{prop}, \ldots, I_{prop}\rangle\). If there exists a model \(\sigma\) with \(\sigma_{prop}\) canonical and \(\sigma \models P\) as in the logic, then we represent this by \(\sigma \models P\), and call \(\sigma_{prop}\) a canonical interpretation sequence on propositions of \(P\). Let \(\Sigma_{\sigma} = \{\sigma \models \vdash_{\sigma}, P\}\) and \(\sigma, \sigma' \in \Sigma_{\sigma}\). We define: (1) \(\sigma_{prop} \subseteq \sigma'\) if and only if \(|\sigma| = |\sigma'|\) and \(I_{prop} \subseteq I_{prop}'\) for all \(0 \leq i \leq |\sigma|\); (2) \(\sigma \subseteq \sigma'\) if and only if \(\sigma_{prop} \subseteq \sigma_{prop}'\); (3) \(\sigma \subseteq \sigma\) if and only if \(\sigma_{prop} \subseteq \sigma_{prop} \land I_{ar} \models P\); (4) \(\sigma \models \sigma'\) if and only if \(\sigma \subseteq \sigma'\).

Since framing destroys monotonicity in MSVL, canonical models are no longer appropriate for framed programs. To deal with this problem, a minimal model theory is developed, by which the temporal semantics of MSVL programs is captured. Let \(I = \langle (\sigma, i, k, j)\rangle\) be a canonical interpretation. Then we have \((\sigma, i, k, j)\models_{m} P\) if and only if \((\sigma, i, k, j)\models P\) and there is no \(\sigma'\) such that \(\sigma' \subseteq \sigma\) and \((\sigma, i, k, j)\models P\). We call \(\models_{m}\) the minimal satisfaction relation. A minimal model of a program \(P\) is a canonical model \(\sigma\) such that \((\sigma, 0, 0, |\sigma|)\models_{m} P\), which is represented by \(\sigma \models_{m} P\). Moreover, the existence of a minimal model for a given framed program is proved.

2) Operational Semantics of MSVL: To capture the executable behaviors of framed programs in an operational way, we present the operational semantics of MSVL based on structure operational semantics in [35, 37]. To this end, new configurations with intervals for MSVL programs are defined. A configuration regarding an MSVL program \(P\) is a quadruple \(c_{p} = (P, \sigma_{p-1}, s_{i}, i)\), where \(P\) is a program, \(\sigma_{p-1} = (s_{0}, \cdots, \)}
Reviews

$s_{i-1}(i>0)$ is a model which records information of an interval, $s_i$ is the current state and $i$ is a counter for counting the number of states in $s_{i-1}$. Further, for $i=0$, let $s_{i-1} = \varepsilon$ be an empty sequence. If a program is terminable and satisfiable, its final configuration arrives at $c_i = (\text{true}, \sigma, \emptyset, |\sigma| + 1)$. For a boolean expression $b$, the configuration is denoted by $c_b = (b, s_{i-1}, s_i, i)$ while for an arithmetic expression $e$, the configuration by $c_e = (e, s_{i-1}, s_i, i)$.

The operational rules are formalized into two categories: one for state reduction and the other for interval reduction. The state reduction is concerned with how to reduce an MSVL program within a state. It comprises evaluation rules of boolean expressions for mapping boolean expressions into a truth value, evaluation rules of arithmetic expressions for mapping arithmetic expressions into an element $c$ in $D$, semantic equivalence rules for rewriting MSVL statements into its normal form, and transition rules within a state for handling concurrent assignments and capturing the minimal model semantics. The interval reduction is about a program executed over an interval and consists of two interval rules to transfer a program from one state to its successors or terminate a program. In addition, the consistency between the operational semantics and the minimal model semantics based on model theory is proved in detail.

3) Axiomatic Semantics of MSVL: For verifying properties of concurrent programs in a unified notation, we establish the axiomatic semantics of MSVL [36, 38], which employs PPTL as the specification language. Let $A, B$ be PPTL formulas and $P$ be an MSVL program. Thus, the correctness assertion can be defined in the form of a variation of Hoare's triple [3] as

$$\langle s_i, A \rangle \ P \langle s_i, B \rangle$$

Therein, $A$ and $s_i = \langle s_1, \ldots, s_k \rangle$ are left-conditions, where $s_i$ represents an interval and implies that $P$ is being deduced at state $s_i$; $B$ and $s_k = \langle s_k, \ldots, s_1 \rangle$ are right-conditions, where $s_k$ indicates the whole interval and program $P$ will be deduced over $\langle s_1, \ldots, s_k \rangle$. Actually, $s_i$ is a prefix of $s_k$. During the deduction, if $s_{i-1}$ satisfies program $P$, then it should satisfy condition $A$ and the final state $s_k$ should satisfy condition $B$.

The axiomatic system can be classified into two sorts: the state deduction and the interval deduction. The state deduction transforms an MSVL program into its normal form within a state. A set of state axioms and inference rules is formalized. Further, the interval deduction is regarding programs deduced over an interval and simultaneously properties verified over the interval. A group of interval axioms and inference rules is defined. In addition, a formal proof of the soundness and relative completeness of the axiomatic system with respect to the operational model of MSVL is given. With the axiomatic system, firstly a concurrent system is modeled with MSVL; secondly its desired properties are specified by PPTL; thirdly whether or not the model satisfies a property can be verified by means of the axioms and inference rules. Thus, modeling, specification and verification can be unified in the same logic PTL.

4.4 Time constraints in MSVL

Real-time systems appear in many safety critical systems and always require some strict quantitative time constraints. To specify and verify real-time systems, we extend MSVL with absolute time constraints and call this real-time extension of MSVL as TMSVL [39]. To do this, explicit time variables and a time duration are needed. Firstly, two time variables $T$ and $T'$ are introduced, which range over non-negative and positive integers respectively. $T$ acts as a global clock to describe time and $T'$ serves as the clock generator to describe time increment. At each state, $T'$'s value stands for the current time and $T'$'s value records the time difference between the current state and the next state. Then a time horizon named time duration can be defined which is related to variable $T$ and used as the time constraints bounded onto a formula. On prefixing an MSVL statement with a time duration, a TMSVL formula is obtained. For example, let $P$ is an MSVL program, then $(t_1, t_2) P$ is a TMSVL program and claims that $P$ is executed over the time duration from $t_1$ to $t_2$.

4.5 Cylinder computation model

Nowadays many-core parallel computing and programming are new challenges to formal specification and verification. In [40], we present a flexible parallel model in temporal logic programming so that it can be used to handle applications of many-core parallel computing. Exactly, based on projection constructs in
• Reviews •

PTL and MSVL, a semantic model for many-core parallel computing, namely cylinder computation model (CCM) is presented. CCM is of a typical form of

$$\phi_1 \text{ov}(l_1) \parallel \phi_2 \text{ov}(l_2) \parallel \cdots \parallel \phi_n \text{ov}(l_n)$$

where each $\phi_i \ (1 \leq i \leq m)$ is an MSVL program whereas each $l_i$ is a sequence expression. With this parallelism, a main time interval is the sequence of fine-grained unit subintervals with length one while several coarse-grained projected intervals over which processes are interpreted are in parallel with the main time interval. This computation model can be viewed as $m$ processes that share one processor and each occupies an execution core cooperating to complete their tasks in a parallel way. Each process progresses in its own speed and communicates with each other at some global states which indicates the coordination among these processes. Sequence expression $l_i$ is used to control and determine the execution points (states) of $\phi_i$. As we can see, parallel construct ( $\parallel$ ) is the main operator in CCM. Thus $\phi_1 \text{ov}(l_1) \parallel \cdots \parallel \phi_n \text{ov}(l_n)$ is endowed with the semantics of many-core parallel computing.

We can make use of the formalism to model the execution of several processes (threads) deployed over a multi-core processor. For example, the interval satisfying a parallel program $\phi_1 \text{ov}(2,3,3,4) \parallel \phi_2 \text{ov}(3,5,3,6) \parallel \phi_3 \text{ov}(2,1,2,3,3,1,5)$ is given in Figure 3. Here $\phi_1, \phi_2$, and $\phi_3$ are executed over three cores on one chip. They are autonomous and communicate only at some specified time points, which are shared by them. To implement CCM programs in the interpreter for MSVL, the operational semantics of CCM is given. Further, a reduction algorithm is designed and implemented based on the operational semantics.

![Figure 3](image)

4.6 Discussion

Owing to abundant temporal operators, MSVL can express multiple intricate temporal relations, such as sequence, concurrency, non-determinacy, and non-termination. Compared with other temporal logic programming languages, such as XYZ/E, TLA, Tempura, MSVL is more practical. The framing operator frame in MSVL elegantly manages the framing problem, namely how to carry values of variables from the current state to the next state, in temporal logic programming languages well, and implements the automatic allocation and release of the memory for variables. Further, MSVL supports various data types [41], such as integers, float, array, list, and pointers. MSVL also allows many-core parallel programming and integrates three functions: modeling, simulation and verification. Besides, both synchronous and asynchronous communication mechanisms are available in MSVL.

5 Formal verification with MSVL and PPTL

MSVL and PPTL have been employed for specification and verification in many realistic applications. In this section, we will review some model checking and theorem proving approaches with them.

5.1 Model checking

Due to the result that PPTL is decidable, PPTL can be used in model checking. Further, since PPTL can describe full regular language, it can express many kinds of properties, such as safety, liveness, interval-sensitive properties and star properties. Some practical model checking techniques for PPTL have been proposed [42—46]. In addition, a unified model checking approach with MSVL and PPTL is put forward [23].

1) Model Checking PPTL based on SPIN: A model checking technique with PPTL is developed based on SPIN [42]. With this method, the system to be verified is firstly modeled as a PROMELA program $M$,
which actually describes a Büchi automata $A_M$. Secondly a property of the system is specified by a PPTL formula $P$. Then $\neg P$ is equivalently transformed to its LNFG, and further transformed to Never Claim, which also represents a Büchi automaton $A_{\neg P}$. Thirdly, the system can be verified automatically in SPIN by computing the product automaton of $A_M$ and $A_{\neg P}$, and then checking whether the words accepted by the product automaton is empty or not. If the words accepted by the product automaton is empty, the system can satisfy the property; otherwise, the system cannot satisfy the property, and a counterexample can be found. Based on this, we implement a model checker SPIN4PPTL.

2) Symbolic Model Checking with PPTL: We present a symbolic model checking algorithm for PPTL in [46] and develop a tool based on SMV. With our method, the system to be verified is modeled as a Kripke structure $M=\langle S, I, R, L \rangle$, while the property of the system to be verified is described by a PPTL formula $P$. To check the satisfiability of a PPTL formula $P$ against the system $M$, the complemented property $\neg P$ is firstly transformed to its normal form and then its LNFG is constructed which contains possible (finite or infinite) models of $P$. Further, the set of states in $M$ that satisfies $\neg P$, namely $\text{Sat}(\neg P)$, can be calculated recursively with respect to the state space and transition relation. In this way, the model checking problem can be reduced to a non-emptiness checking problem of the intersection set of $\text{Sat}(\neg P)$ and the initial state set $I$. Since both the state space and transition relation are represented symbolically using boolean functions, the above operations can be implemented with efficient graph algorithms operated on ROBDDs [47].

3) Bounded Model Checking with PPTL: We give the bounded model checking approach with PPTL in [45]. To this end, firstly the bounded semantics of PPTL is defined, and some lemmas and theorems for building the relationship between bounded model checking and basic model checking are proved. With these basic theories, bounded model checking with PPTL can be reduced into SAT problem. Given a finite state transition system $M$ (a Kripke structure or NFG), the property of the system in terms of a PPTL formula $f$, and a upper bound $k$ (an integer), then the procedure of bounded model checking can be described as an iterative process for constructing a proposition formula $[M, \neg f]_i = M \land X_i$ and determining its satisfiability with SAT solvers. Thereinto, $i$ is the step increment from 0 to $k$; $M_i$ is a propositional formula that constrains the sub-interval $\langle s_0, \cdots, s_i \rangle$ to be a valid interval starting from an initial state; $X_i$ is a propositional formula that constrains the interval $\sigma$ to satisfy $\neg f$ with bound $i$. When $[M, f]_i = M \land X_i$ is satisfiable, the model $M$ does not satisfy the property $f$ and a counterexample is exhibited. Otherwise, if $i < k$ the value of $i$ is increased and the process is repeated; or else, $f$ is satisfied by $M$ against the upper bound $k$. We also implement a bounded model checker for PPTL based on NuSMV.

4) Abstract Model Checking with PPTL: In abstract model checking, a counterexample in an abstract model may not be a real counterexample in the concrete model. Accordingly, the abstract model needs to be further refined. Clarke et al. proposes the counterexample guided abstraction and refinement method [9] with abstraction performed by selecting a set of variables that are insensitive to the desired property to be invisible. If a counterexample is checked to be spurious, a set of variables are made visible to refine the abstract model. With this method, to find the coarsest (or smallest) refined model is NP-hard [48]. Further, it is important to find a small set of variables in order to keep the size of the abstract state space smaller, which is also NP-hard [49]. We present a novel refinement method [43,50] for this abstraction approach. When a failure state is detected, instead of selecting some invisible variables to be visible, extra variables are added to the abstract model for refinement. In this way, the NP-hard state separation problem can be avoided and a smaller refined abstract model can also be obtained.

In abstract model checking, how to check whether or not a reported counterexample is spurious is a key problem. The algorithm SPLITPATH [48] is for this purpose. To determine whether or not a state, say $\tilde{s}$, is a failure state, SPLITPATH relies on the prefix of the counterexample $\tilde{s}_0, \tilde{s}_1, \cdots, \tilde{s}_r$. This brings in a polynomial number of unwinding of the loop in an infinite counterexample. We give a formal definition for spurious path and put forward a new technique for checking spurious counterexamples [44,51]. Within our approach, whether or not a counterexample is spurious still depends on the existence of failure states in the counterexample. However, instead of the prefix, examining whether or not a state $\tilde{s}$ is a failure state
Reviews

is only up to \( s_i \)'s pre- and post- states in the counterexample. Based on this, for an infinite counterexample, the polynomial number of unwinding of the loop can be avoided. Further, the algorithm can be easily improved by detecting the heaviest failure state such that a number of model checking iterations can be saved in the whole abstract-refinement loop. In addition, the algorithm can be naturally paralleled. Thus, combing this new algorithm for checking spurious counterexamples and the new abstraction refinement algorithm, an abstract model checking approach for PPTL and an abstract model checker can be acquired.

5) A Unifying Model Checking Approach with PTL

We give a unified model checking approach with PTL based on LNFG [23] and develop a tool MSV. With this method, a system is first modeled as an MSVL program \( M \). Secondly, a property of the system is specified as a PPTL formula \( P \). Since both PPTL and MSVL are subsets of PTL, model checking can be unified in one logic framework PTL. Further, to check whether or not \( M \) satisfies \( P \) amounts to proving whether or not \( M \land \neg P \) is satisfiable. As finite state programs in MSVL are equivalent to PPTL formulas, the model checking problem can be translated into a satisfiability problem in PPTL. We have proved that PPTL is decidable and given a decision procedure in [24, 26]. With this procedure, a PPTL formula is satisfiable if and only if there is a valid finite or infinite path in its LNFG. Therefore, the problem of checking whether or not \( M \) satisfies \( P \) is eventually translated to the problem of checking whether or not the LNFG of \( M \land \neg P \) contains a valid path.

Based on the above analysis, a model checking algorithm can be given as follows: (1) modeling the system as a MSVL program \( M \) and specifying the property of the system as a PPTL formula \( P \); (2) constructing the LNFG of \( M \land \neg P \); (3) checking the LNFG to find out a counterexample if the LNFG contains valid paths; otherwise, outputting 'satisfiable' message. Actually, during building the LNFG of \( M \land \neg P \), we can transform \( M \) and \( \neg P \) into their normal forms separately and then construct their conjunctions. In this way, a more effective recursive algorithm can be obtained. More details can refer to [23].

5.2 Theorem proving

As a complement for model checking, we also investigate the theorem proving approach with PPTL and MSVL. The theorem proving method with MSVL proceeds exactly on the axiomatic semantics of MSVL, which has been briefly introduced in Section 4.5. At present, it has been applied in verifying concurrent and distributed systems, such as distributed mutual exclusion algorithms [38]. For PPTL, a sound and complete proof system has been established [52]. It has been successfully used in the verification of real-time systems [53] and hardware systems. The proof system for PPTL consists of a set of axioms and inference rules. Further, some useful theorems within the axiomatization are also derived. In addition, the axiom system is proved to be sound and complete. With this proof system, both a system and a desired property are specified by PPTL formulas, say \( S \) and \( P \). The system satisfies the property if and only if we can find a formal proof of \( \vdash S \rightarrow P \) using axioms and inference rules. A proof assistant for PPTL based on PVS has been recently developed.

Note that the proof system for PPTL not only can deal with the chop, chop-star and the projection operators, but also can handle a new projection-plus operator \( \bigoplus \text{prj} \). In a formula \( (P_1, \ldots, P_i, \ldots, P_j, \bigoplus, \ldots, P_n) \text{prj} Q \), the sequence \( P_i, \ldots, P_j \) is referred to as the iterative part. The difference between the projection and projection-plus constructs is that the formulas \( P_i, \ldots, P_n \text{in} (P_1, \ldots, P_n) \text{prj} Q \) are interpreted sequentially with each formula interpreted only once while the iterative part in \( (P_1, \ldots, P_i, \ldots, P_j, \bigoplus, \ldots, P_n) \text{prj} Q \) can be interpreted repeatedly (at least once). Moreover, the new projection-plus operator can subsume chop, chop-star and the original projection (\( \text{prj} \)) operators.

6 A supporting toolkit MSV

Based on the above theories and approaches, a supporting toolkit MSV for modeling, simulation and verification with MSVL and PPTL is implemented. With MSV, we can edit MSVL programs, model, simulate and verify systems with MSVL and PPTL, as well as automatically translate C, Verilog, VHDL programs and workflow nets into MSVL programs. The framework of MSV is showed in Figure 4. It mainly consists of 5 parts: an interpreter, a modeling tool, a simulation tool, verification toolkits and translators X2MSVL from other high-level languages.
6.1 Interpreter
The interpreter for MSVL aims at editing, debugging, interpreting and executing MSVL programs. Similar to other tools, such as Visual C++, Eclipse, MSV can support syntax highlighting and errors indicating, which facilitates editing and debugging. The interpreter is mainly composed of 8 modules.

- Editor; editing MSVL programs.
- Lexical Analyzer; checking whether or not there are lexical errors in MSVL programs by means of Flex. If there are no lexical errors in programs, a token stream is obtained; otherwise, errors and their locations are reported.
- Parser; checking whether or not the grammar of MSVL programs is correct using Bison. If the grammar is correct, syntax trees of programs and properties are generated.
- Data Input; inputting some required data and setting some control parameters by users.
- Reduction; reducing the syntax tree of a program (res. a property) into two parts, i.e. the present components and future components. Then the present components are executed in the current state. If this is successful, the syntax tree of the program is imported for the next step and future components will be reduced; otherwise, the syntax tree is traversed again until it is reduced successfully.
- Symbol Table Module; storing global variables in each state using a symbol table. At the end of each state, the symbol table is updated and some non-framed variables are cleared. When the reduction of the program terminates, the symbol table will be deleted.
- Present and Remains Module; storing the two parts of the syntax trees at each state.
- Output; displaying the details of the reduction at each state and the final result.

6.2 Modeling tool
With the modeling tool, given an MSVL program $P$, its state space of $P$ can be generated and output. Since the state space of an MSVL program are implicitly presented as its NFG, the modeling tool displays the NFG of a program. In this way, we can obtain the model of systems. Based on the state space, we can analyze the properties of the system, such as reachability, deadlock. Actually, the modeling tool is based on the interpreter and a model of $P$ is produced along with the reduction of $P$ in an interpreter.

6.3 Simulation tool
The simulation tool can simulate one execution path of the NFG of the system and output all the states over the path according to the minimal model semantics of MSVL. Within the simulation tool, users can set the maximal numbers of the states in a path. According to the path, we can track errors and find out the locations of the errors.
6.4 Verification toolkits

The verification toolkits contains a series of model checkers and two theorem provers. The theorem provers are composed of a proof assistant PVS4PPTL for the axiomatic system of PPTL and a theorem prover PVS4MSVL for the axiomatic system of MSVL. Both of them are based on PVS. To implement them, first MSVL and PPTL are expressed by the specification language of PVS, which enables PVS to recognize MSVL and PPTL correctly. Then the axiomatic systems for PPTL and MSVL and the theorem to be verified are specified. Further, the proof commands of PVS are input for invoking the PVS prover to deduce the theorem.

The model checkers consists of a unified model checker, a partial-order model checker PMC4PPTL, a symbolic model checker SMC4PPTL and a bounded model checker BM-C4PPTL, and an abstract model checker AMC4PPTL. They are developed on the techniques above. The partial-order model checker PMC4PPTL is based on the model checker SPIN and extends SPIN to deal with PPTL. Both the symbolic and bounded model checkers are built on SMV. The abstract model checker AMC4PPTL modifies CPAchecker [54] with the new abstract refinement algorithm and the algorithm for checking spurious counterexamples as well as some mechanisms for PPTL. The unified model checker employs MSVL as the modeling language and PPTL as the specification language, which facilitates verifying in the same notation.

6.5 Translators

C language is a general imperative language and VHDL and Verilog are the most popular hardware description languages. The three languages have been widely used in practice. To guarantee the correctness of C/VHDL/Verilog programs, we propose a method with MSVL. To do this, a program written in C/VHDL/Verilog is first translated to an MSVL program, and then the task is changed to verify MSVL programs. In this way, we can verify C/VHDL/Verilog programs through the verification tools for MSVL. In order to automatically translate C/VHDL/Verilog programs into MSVL programs, three tools C2MSVL, Verilog2MSVL and VHDL2MSVL are developed respectively for C, Verilog and VHDL. They proceed as follows; first C/VHDL/Verilog programs are preprocessed via adding the header files into the current programs; then based on the syntax rules of C/VHDL/Verilog, a lexical and grammar analysis is made and a syntax tree is generated; further, a series of translation rules from C/VHDL/Verilog to MSVL are applied to the syntax tree; finally, a corresponding MSVL program is obtained.

As a subset of Petri nets, Workflow Nets (WFN) provide not only a graph representation and but also a mathematical semantics. To verify a WFN, we also put forward a technique by translating workflow nets to MSVL. To this end, firstly condition and statement annotations are inserted into a workflow net, which generates the annotation workflow net. Secondly, the regular structures in the annotation workflow net are reduced repeatedly with a set of translation rules until that only one transition is included in the annotation workflow. Thirdly, some necessary MSVL statements, such as frame statement and process definition statement, are added. As a result, an MSVL program corresponding to the workflow net is acquired. An automatic translation tool named PN2MSVL from workflow net to MSVL is implemented.

6.6 Discussion

Compared with other famous tools, such as SPIN, NuSMV and CPAchecker, our toolkit MSV has some differences and advantages, which are shown in Table 6:

<table>
<thead>
<tr>
<th>Specification language</th>
<th>The toolkit MSV</th>
<th>SPIN</th>
<th>NuSMV</th>
<th>CPAchecker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Properties</td>
<td>full regular, interval sensitive</td>
<td>safety, liveness</td>
<td>safety, liveness</td>
<td>—</td>
</tr>
<tr>
<td>Modelling language</td>
<td>MSVL, C, Verilog, VHDL, Promela, SMV, WFN</td>
<td>Promela</td>
<td>SMV</td>
<td>C</td>
</tr>
<tr>
<td>Improved MC techniques</td>
<td>abstract, bounded, symbolic, partial-order</td>
<td>abstract, bounded, symbolic, partial-order</td>
<td>symbolic, bounded</td>
<td>abstract</td>
</tr>
<tr>
<td>Languages</td>
<td>a unified language</td>
<td>three languages</td>
<td>three languages</td>
<td>three languages</td>
</tr>
</tbody>
</table>

Table 6 The comparison between MSV and other verifiers
• Reviews •

• MSV takes PPTL as the specification language while other tools employ traditional LTL and CTL to specify properties. As PPTL can express full regular language, PPTL can express not only the safety and liveness properties, but also interval sensitive properties and full regular properties. As a result, MSV can support full regular properties whereas other tools cannot.

• MSV accepts many types of languages as its modeling languages, which makes it more practical. MSVL is its main modeling language. Further, MSV integrates a number of verification tools, such as PMC4PPTL, SMC4PPTL, BMc4PPTL, which enables Promela and SMV also to be its modeling languages. Moreover, due to the translators X2MSVL, C, VHDL, Verilog and WFN can be accepted by MSV.

• In SPIN, NuSMV and CPAchecker, modeling, specification and verification are performed with different languages, which is not convenient and takes a burden for complex transformations between different formalisms. On the contrary, in MSV, since both MSVL and PPTL are subsets of PTL, modeling, specification and verification can be conducted within the same language PTL.

• MSV takes many model checking approaches, such as abstract, bounded, symbolic and partial-order model checking techniques, which greatly reduces the state space and improves the efficiency of model checking. Further, MSV also combines the methods of model checking and theorem proving.

7 Conclusion

In this paper, we reviewed some significant theory and application results of PPTL and MSVL. First, we present the decidability, complexity and expressiveness results of PPTL. Then we demonstrate some achievements on MSVL involving basic statements, asynchronous communication, formal semantics, time constraints and cylinder computation model. Further, we give some model checking and theorem proving methods with PPTL and MSVL. Besides, we briefly introduce the practical toolkit MSV.

PPTL and MSVL have been successfully applied in specifying and verifying many practical fields, such as C programs, Verilog/VHDL programs, virtual memory management, distributed mutual exclusion protocols, multi-core computation and hardware systems. Although basic issues on PPTL and MSVL have been solved well, there still exist some aspects to be considered in the future. To combat the state explosion problems in model checking, we will investigate the improved model checking techniques for MSVL, such as partial-order/symbolic/bounded/abstract model checking with MSVL. At present, the theorem prover for PPTL and MSVL are based on PVS. However, the degree of automaton is still not high. Thus, we will improve the degree of automaton and attempt to develop another theorem prover for PPTL and MSVL based on other theorem provers, like Coq. Moreover, although the toolkit MSV works well in practice, it still needs to be optimized to achieve better performance.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (Grant Nos. 61133001, 61272117, 61202038, 61322202, 61420106004 and 91418201).

References

• Reviews •


[23] Duan Z, Tian C. A unified model checking approach with projection temporal logic. ICFEM, 2008; 167—188.


Tian Cong received the BS, MS and PhD degrees in computer science from Xidian University, China in 2004, 2007 and 2009, respectively. She is a professor at the Institute of Computing Theory and Technology (ICTT), Xidian University, China. She was a visiting postdoctoral researcher in Hosei University, Japan from 2010 to 2011. Her research interests include theories in model checking, temporal logics and automata, formal verification of software systems, and software engineering. She is a member of the IEEE.

Duan Zhenhua received the PhD degrees in computer science from both Newcastle University and the University of Sheffield, UK. He is a professor at the Institute of Computing Theory and Technology (ICTT), Xidian University, China. His research interests include model checking, temporal logics, formal verification of software systems, and temporal logic programming. He is a senior member of the IEEE.