A Unified Model Checking Approach with Projection Temporal Logic*

Zhenhua Duan and Cong Tian

Institute of Computing Theory and Technology, Xidian University,
Xi'an, 710071, P.R. China
{zhhduan,ctian}@mail.xidian.edu.cn

Abstract. This paper presents a unified model checking approach with Projection Temporal Logic (PTL) based on Normal Form Graphs (NFGs). To this end, a Modeling, Simulation and Verification Language (MSVL) is defined based on PTL. Further, normal forms and NFGs for MSVL programs and Propositional PTL (PPTL) formulas are defined. The finiteness for NFGs of MSVL programs is proved in details. Moreover, by modeling a system with an MSVL program p, and specifying the desirable property of the system with a PPTL formula ϕ, whether or not the system satisfies the property (whether or not $p \to \phi$ is valid) can equivalently be checked by evaluating whether or not $\neg(p \to \phi) \equiv p \wedge \neg\phi$ is unsatisfiable. Finally, the satisfiability of a formula in the form of $p \wedge \neg\phi$ is checked by constructing the NFG of $p \wedge \neg\phi$, and then inspecting whether or not there exist paths in the NFG.

1 Introduction

Verification and testing are basic techniques to validate systems [11,23,24] at the present. Model checking is an automatic verification approach based on model theory. To verify whether or not a system meets a property, the system is modeled as a finite transition system or automaton M, and the property is specified by a temporal logic formula p. Then a model checking procedure is employed to check whether or not $M \models p$. The advantage of model checking is that the verification can be done automatically. However, it suffers from the state explosion problem. Also, it is less suitable for data intensive applications since the treatment of data usually produces infinite state spaces [8]. Two successful model checking tools are SPIN [7] and SMV [8].

The state explosion problem is typically caused by models growing exponentially in the number of parallel components or data elements of an argument system. This observation has led to a number of techniques for fighting this problem. The most rigorous approaches are compositional ones [12,17,18,19], trying to avoid the problem in a divide and conquer fashion. Partial order methods

* This research is supported by the NSFC Grant No. 60433010, and Defense Pre-Research Foundation of China, Grant No. 51315050105.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 167–186, 2008.
© Springer-Verlag Berlin Heidelberg 2008

limit the size of the models representation by suppressing unnecessary interleavings, which typically arise as a result of the serialization during the model construction of concurrent systems [13,14,15,16].

The most significant improvement to model checking is made by Symbolic Model Checking (SMC) [8,9,10] and Bounded Model Checking (BMC) [20]. In SMC, sets of states are represented implicitly using boolean functions which can be manipulated efficiently with Reduced Ordered Binary Decision Diagram (ROBDD, or BDD for short) [21]. As a result, SMC allows a polynomial system representation but may explode in the course of the model checking process. The combination of SMC with BDDs pushed the barrier to systems with 10^{20} states and more [10]. However, the bottleneck of SMC methods is the amount of memory that is required for storing and manipulating BDDs. Although numerous techniques such as decomposition, abstraction, and various reductions have been proposed through the years to overcome this problem, full verification of many designs is still beyond the capacity of BDD based SMC.

The basic idea in BMC is to search for a counterexample in executions whose length is bounded by some integer k [20]. If no bug is found then we increases k until either a bug is found, the problem becomes intractable, or some pre-known upper bound is reached. The BMC problem can be efficiently reduced to a propositional satisfiability problem, and can therefore be solved by SAT methods rather than BDDs. Experiments have shown that it can solve many systems that cannot be solved by BDD-based techniques. However, BMC does not solve the complexity problem of model checking since it still relies on an exponential procedure and hence is limited in its capacity. BMC also has the disadvantage of not being able to prove the absence of errors.

In this article, we present a unified model checking approach with Projection Temporal Logic (PTL) based on Normal Form Graphs (NFGs) [5]. With this method, a system is first modeled as P using a modeling, simulation and verification language called MSVL which is a subset of PTL [2,25] and an extension of Framed Tempura [6]. Thus, P is a non-deterministic program of MSVL and also a formula of PTL. Second, a property of the system is specified by a formula ϕ of Propositional PTL (PPTL) [2,5]. To check whether or not P satisfies ϕ amounts to proving $\models P \rightarrow \phi$. It turns out equivalently to prove $\not\models P \wedge \neg\phi$. Thus, we translate the model checking problem into a satisfiability problem in PPTL since finite state programs in MSVL are equivalent to PPTL formulas (see Appendix C). As a result, we have proved that PPTL is decidable and given a decision procedure in [5]. With this procedure, a PPTL formula is satisfiable if and only if there is a valid path in its NFG. Therefore, the problem of checking whether or not P satisfies ϕ is eventually translated to the problem of checking whether or not the NFG of $P \wedge \neg\phi$ contains a valid finite or infinite path. If not, the property is verified otherwise a valid path of the NFG determines a counterexample. Based on the above analysis, a model checking algorithm can be given as follows: (1) modeling the system as program P in MSVL and specifying the property of the system as a PPTL formula ϕ; (2) constructing the NFG of $P \wedge \neg\phi$; (3) checking the NFG to find out a counterexample if the NFG contains valid paths

otherwise output 'satisfied' message. However, a further analysis tells us that a more effective recursive algorithm can be given since we can transform P and $\neg\phi$ into their normal forms separately and the conjunction of P and $\neg\phi$ can be reduced to the form in $P_e \wedge \phi_e \wedge \varepsilon \vee \bigcirc(P' \wedge \neg\phi')$. Thus, the NFG of the original formula $P \wedge \neg\phi$ can recursively be constructed.

Our method has some advantages. For instance, (1) the model and property of a system can be written in the same logic; (2) the model checking algorithm is relies on constructing the NFG of a PPTL formula; during the construction, when a valid finite or infinite path has been constructed the algorithm immediately stops since we do not need to construct the whole NFG of the formula if we do not expect to have all counterexamples; (3) the existing SAT procedure can be reused to check the satisfaction of the state formulas with the present components of a normal form; (4) the expressiveness of PPTL is more powerful than Propositional Linear TL (PLTL) since we have proved that the expressiveness of PPTL is equivalent to the full regular expression [22] but that of PLTL equals star free regular expression [26,27]. However, in the worst case, our model checking approach does not solve the complexity problem of model checking since it still relies on an exponential procedure and hence is limited in its capacity.

This paper is organized as follows. In the following section, the syntax, semantics and some logic laws of PTL are presented. In Section 3, the language MSVL is formalized, the normal form and NFG of MSVL are defined, and finiteness for NFGs of MSVL is proved. Correspondingly, as a property specification language, the syntax, semantics, normal form and NFGs of PPTL formulas are briefly introduced in Section 4. In Section 5, the unified model checking approach with PTL based NFGs is presented. Further, an example is given to show how the model checking algorithm and the developed supporting tools work. Finally, conclusions are drawn in Section 6.

2 Projection Temporal Logic

Our underlying logic is Projection Temporal Logic [3,2], it is an extension of Interval Temporal Logic (ITL) [4]. Let Π be a countable set of propositions, and V be a countable set of typed static and dynamic variables. $B = \{true, false\}$ represents the boolean domain and D denotes all the data we need. The terms e and formulas p of the logic are given by the following grammar:

$$e ::= v \mid \bigcirc e \mid \circleddash e \mid beg(e) \mid end(e) \mid f(e_1, ..., e_n)$$

$$p ::= \pi \mid e_1 = e_2 \mid P(e_1, ..., e_n) \mid \neg p \mid p_1 \wedge p_2 \mid \exists x : p \mid \bigcirc p \mid (p_1, ..., p_m) prj\ p \mid p^+$$

where $\pi \in \Pi$ is a proposition, and v a dynamic variable or a static variable. In $f(e_1, \ldots, e_n)$ and $P(e_1, \ldots, e_n)$, f is a function and P is a predicate. It is assumed that the types of the terms are compatible with those of the arguments of f and P. A formula (term) is called a state formula (term) if it does not contain any temporal operators (i.e. $\bigcirc$, $\circleddash$, $beg(.)$, $end(.)$ and prj); otherwise it is a temporal formula (term).

A state s is a pair of assignments (I_v, I_p) where for each variable $v \in V$ defines $s[v] = I_v[v]$, and for each proposition $\pi \in \Pi$ defines $s[\pi] = I_p[\pi]$. $I_v[v]$ is a value in D or nil (undefined), whereas $I_p[\pi] \in B$. An interval $\sigma = \langle s_0, s_1, \dots \rangle$ is a non-empty (possibly infinite) sequence of states. The length of σ, denoted by $|\sigma|$, is defined as ω if σ is infinite; otherwise it is the number of states in σ minus one. To have a uniform notation for both finite and infinite intervals, we will use extended integers as indices. That is, we consider the set N_0 of non-negative integers and ω, $N_\omega = N_0 \cup \{\omega\}$, and extend the comparison operators, $=, <, \leq$, to N_ω by considering $\omega = \omega$, and for all $i \in N_0$, $i < \omega$. Moreover, we define $\preceq$ as $\leq -\{(\omega, \omega)\}$. With such a notation, $\sigma_{(i..j)}$ $(0 \leq i \leq j \leq |\sigma|)$ denotes the sub-interval $< s_i, ..., s_j >$ and $\sigma^{(k)}$ $(0 \leq k \leq |\sigma|)$ denotes $< s_k, ..., s_{|\sigma|} >$. The concatenation of σ with another interval (or empty string) σ' is denoted by $\sigma \cdot \sigma'$. To define the semantics of the projection operator we need an auxiliary operator for intervals. Let $\sigma = \langle s_0, s_1, \dots \rangle$ be an interval and $r_1, \dots, r_h$ be integers $(h \geq 1)$ such that $0 \leq r_1 \leq r_2 \leq \dots \leq r_h \leq |\sigma|$. The projection of σ onto $r_1, \dots, r_h$ is the interval (called projected interval), $\sigma \downarrow (r_1, \dots, r_h) = \langle s_{t_1}, s_{t_2}, \dots, s_{t_l} \rangle$, where $t_1, \dots, t_l$ is obtained from $r_1, \dots, r_h$ by deleting all duplicates. For example,

$$\langle s_0, s_1, s_2, s_3, s_4 \rangle \downarrow (0, 0, 2, 2, 2, 3) = \langle s_0, s_2, s_3 \rangle$$

An interpretation for a PTL term or formula is a tuple $\mathcal{I} = (\sigma, i, k, j)$, where $\sigma = \langle s_0, s_1, \dots \rangle$ is an interval, i and k are non-negative integers, and j is an integer or ω, such that $i \leq k \leq j \leq |\sigma|$. We use (σ, i, k, j) to mean that a term or formula is interpreted over a subinterval $\sigma_{(i..j)}$ with the current state being s_k. For every term e, the evaluation of e relative to interpretation $\mathcal{I} = (\sigma, i, k, j)$ is defined as $\mathcal{I}[e]$, by induction on the structure of a term, where v is a variable and $e_1, \dots, e_m$ are terms.

$$\begin{aligned}
\mathcal{I}[v] = s_k[v] &= I_v^k[v] = I_v^i[v], \text{ if } v \text{ is a static variable.} \\
\mathcal{I}[v] = s_k[v] &= I_v^k[v], \text{ if } v \text{ is a dynamic variable.} \\
\mathcal{I}[f(e_1, \dots, e_m)] &= \begin{cases} f(\mathcal{I}[e_1], \dots, \mathcal{I}[e_m]), \text{ if } \mathcal{I}[e_h] \neq nil \text{ for all } h \\ nil, \text{otherwise} \end{cases} \\
\mathcal{I}[\bigcirc e] &= \begin{cases} (\sigma, i, k+1, j)[e], \text{ if } k < j \\ nil, \text{ otherwise} \end{cases} \\
\mathcal{I}[\ominus e] &= \begin{cases} (\sigma, i, k-1, j)[e], \text{ if } i < k \\ nil, \text{ otherwise} \end{cases} \\
\mathcal{I}[beg(e)] &= (\sigma, i, i, j)[e] \\
\mathcal{I}[end(e)] &= \begin{cases} (\sigma, i, j, j)[e], \text{ if } j \neq \omega \\ nil, \text{ otherwise} \end{cases}
\end{aligned}$$

The satisfaction relation for formulas $\models$ is inductively defined as follows.

1. $\mathcal{I} \models \pi$ if $s_k[\pi] = I_p^k[\pi] = true$.
2. $\mathcal{I} \models P(e_1, \dots, e_m)$ if $P(\mathcal{I}[e_1], \dots, \mathcal{I}[e_m]) = true$ and $\mathcal{I}[e_h] \neq nil$, for all h.
3. $\mathcal{I} \models e = e'$ if $\mathcal{I}[e] = \mathcal{I}[e']$.
4. $\mathcal{I} \models \neg p$ if $\mathcal{I} \not\models p$.
5. $\mathcal{I} \models p \wedge q$ if $\mathcal{I} \models p$ and $\mathcal{I} \models q$.

6. $\mathcal{I} \models \bigcirc p$ if $k < j$ and $(\sigma, i, k+1, j) \models p$.
7. $\mathcal{I} \models \exists x : p$ if for some interval σ' which has the same length as σ, $(\sigma', i, k, j) \models p$ and the only difference between σ and σ' can be in the values assigned to variable x.
8. $\mathcal{I} \models (p_1, \ldots, p_m)\, prj\, q$ if there exist integers $k = r_0 \leq r_1 \leq \ldots \leq r_m \preceq j$ such that $(\sigma, i, r_0, r_1) \models p_1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$ (for $1 < l \leq m$), and $(\sigma', 0, 0, |\sigma'|) \models q$ for one of the following σ':
   (a) $r_m < j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_m) \cdot \sigma_{(r_m+1..j)}$
   (b) $r_m = j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_h)$ for some $0 \leq h \leq m$.
9. $\mathcal{I} \models p^+$ if there are $k = r_0 \leq r_1 \leq \ldots \leq r_{n-1} \preceq r_n = j$ $(n \geq 1)$ such that $(\sigma, i, r_0, r_1) \models p$ and $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$ for all $1 < l \leq n$.

A formula p is satisfied by an interval σ, denoted by $\sigma \models p$, if $(\sigma, 0, 0, |\sigma|) \models p$; a formula p is satisfiable if $\sigma \models p$ for some σ. A formula p is valid, denoted by $\models p$, if $\sigma \models p$ for all σ. A formula p is lec-formula if $(\sigma, k, k, j) \models p \Leftrightarrow (\sigma, i, k, j) \models p$ for any interpretation (σ, i, k, j).

The abbreviations $true$, $false$, $\wedge$, $\rightarrow$ and $\leftrightarrow$ are defined as usual. In particular, $true \stackrel{\text{def}}{=} P \vee \neg P$ and $false \stackrel{\text{def}}{=} P \wedge \neg P$ for any formula P. Also we have the following derived formulas:

$$
\begin{array}{rcl@{\qquad}rcl}
empty & \stackrel{\text{def}}{=} & \neg \bigcirc true & more & \stackrel{\text{def}}{=} & \neg empty \\
len(0) & \stackrel{\text{def}}{=} & empty & len(n) & \stackrel{\text{def}}{=} & \bigcirc len(n-1), n \geq 1 \\
skip & \stackrel{\text{def}}{=} & len(1) & \odot P & \stackrel{\text{def}}{=} & empty \vee \bigcirc P \\
P;\ Q & \stackrel{\text{def}}{=} & (P,\ Q)\ prj\ empty & \Diamond P & \stackrel{\text{def}}{=} & true\ ;\ P \\
\Box P & \stackrel{\text{def}}{=} & \neg \Diamond \neg P & p^* & \stackrel{\text{def}}{=} & empty \vee p^+
\end{array}
$$

Some useful logic laws of PTL can be found in Appendix A and their proofs can be found in [5,6].

3 Modeling, Simulation and Verification Language

The Language MSVL is a subset of Projection Temporal Logic with framing technique, and an extension of Framed Tempura [6]. It can be be used for the purpose of modeling, simulation and verification of software and hardware systems.

3.1 Framing

Framing is concerned with the persistence of the values of variables from one state to another. Intuitively, the framing operation on variable x, denoted by $frame(x)$, means that variable x always keeps its old value over an interval if no assignment to x is encountered. For the definition of $frame$ operator, a new assignment called *a positive immediate assignment* is defined as

$$x \Leftarrow e \stackrel{\text{def}}{=} x = e \wedge p_x$$

where p_x is an atomic proposition associated with state (dynamic) variable x, and notice that p_x cannot be used for other purpose. To identify an occurrence of an assignment to a variable, say x, we make use of a flag called the assignment flag, denoted by a predicate $af(x)$; it is true whenever an assignment of a value to x is encountered, and false otherwise. The definition of the assignment flag is $af(x) \stackrel{\text{def}}{=} p_x$, for every variable x. There are state framing (lbf) and interval framing ($frame$) operators. Intuitively, when a variable is framed at a state, its value remains unchanged if no assignment is encountered at that state. A variable is framed over an interval if it is framed at every state over the interval.

$$\begin{aligned} lbf(x) &\stackrel{\text{def}}{=} \neg af(x) \rightarrow \exists b : (\ominus x = b \wedge x = b) \\ frame(x) &\stackrel{\text{def}}{=} \Box(more \rightarrow \bigcirc lbf(x)) \end{aligned}$$

where b is a static variable.

3.2 The MSVL Language

The arithmetic expression e and boolean expression b of MSVL are inductively defined as follows:

$$e ::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_0 \ op \ e_1 (op ::= + \mid - \mid * \mid \backslash \mid mod)$$

$$b ::= true \mid false \mid e_0 = e_1 \mid e_0 < e_1 \mid \neg b \mid b_0 \wedge b_1$$

where n is an integer and x is a variable. The elementary statements in MSVL are defined as follows.

Termination:	$empty$
Assignment:	$x = e$
P-I-Assignment:	$x \Leftarrow e$
State Frame:	$lbf(x)$
Interval Frame:	$frame(x)$
Conjunction:	$p \wedge q$
Selection:	$p \vee q$
Next:	$\bigcirc p$
Always:	$\Box p$
Conditional:	$if\ b\ then\ p\ else\ q \stackrel{\text{def}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$
Exists:	$\exists x : p$
Projection:	$(p_1, \ldots, p_m)\ prj\ p$
Sequence:	$p\ ; q$
While:	$while\ b\ do\ p \stackrel{\text{def}}{=} (p \wedge b)^* \wedge \Box(empty \rightarrow \neg b)$
Parallel:	$p \parallel q \stackrel{\text{def}}{=} (p \wedge (q; true)) \vee (q \wedge (p; true))$
Await:	$await(b) \stackrel{\text{def}}{=} (frame(x_1) \wedge ... \wedge frame(x_h)) \wedge \Box(empty \leftrightarrow b)$ where $x_i \in V_b = \{x \mid x \text{ appears in } b\}$

where x denotes a variable, e stands for an arbitrary arithmetic expression, b a boolean expression, and $p_1, \ldots, p_m$, p and q stand for programs of MSVL. The

assignment $x = e$, positive immediate assignment $x \Leftarrow e$, $empty$, $lbf(x)$, and $frame(x)$ are basic statements and the others are composite ones.

The assignment $x = e$ means that the value of variable x is equal to the value of expression e. Positive immediate assignment $x \Leftarrow e$ indicates that the value of x is equal to the value of e and the assignment flag for variable x, p_x, is $true$. Statements of $if\ b\ then\ p\ else\ q$ and $while\ b\ do\ p$ are the same as that in the conventional imperative languages. The next statement $\bigcirc p$ means that p holds at the next state while $\Box p$ means that p holds at all the states over the whole interval from now. $p \wedge q$ means that p and q are executed concurrently and share all the variables during the mutual execution. $p \vee q$ means p or q are executed. $empty$ is the termination statement meaning that the current state is the final state of the interval over which the program is executed. The sequence statement $p; q$ means that p is executed from the current state to its termination while q will hold at the final state of p and be executed from that state. The existential quantification $\exists x : p$ intends to hide the variable x within the process p. $lbx(x)$ means the value of x in the current state equals to value of x in the previous state if no assignment to x occurs, while $frame(x)$ indicates that the value of variable x always keeps its old value over an interval if no assignment to x is encountered. Different from the conjunction statement, the parallel statement allows both the processes to specify their own intervals. e.g., $len(2) \| len(3)$ holds but $len(2) \wedge len(3)$ is obviously false. Projection can be thought of as a special parallel computation which is executed on different time scales. The projection $(p_1, \ldots, p_m)\ prj\ q$ means that q is executed in parallel with $p_1, \ldots, p_m$ over an interval obtained by taking the endpoints of the intervals over which the $p_i's$ are executed. In particular, the sequence of $p_i's$ and q may terminate at different time points. Finally, $await\ b$ does not change any variable, but waits until the condition b becomes $true$, at which point it terminates.

Further, the following derived statements are useful in practice.

Multiple Selection:	$OR_{k=1}^{n} \stackrel{\text{def}}{=} p_1 \vee p_2 \vee ... \vee p_n$
Conditional:	$if\ b\ do\ p \stackrel{\text{def}}{=} if\ b\ do\ p\ else\ empty$
When:	$when\ b\ do\ p \stackrel{\text{def}}{=} await(b); p$
Guarded Command:	$b_1 \rightarrow p_1 \Box ... \Box b_n \rightarrow p_n \stackrel{\text{def}}{=} OR_{k=1}^{n}(when\ b_k\ do\ p_k)$
Repeat:	$repeat\ p\ until\ c \stackrel{\text{def}}{=} p; while\ \neg c\ do\ p$

3.3 Normal Forms and NFGs of MSVL

Definition 1. A program q in MSVL is in normal form if

$$q \stackrel{\text{def}}{=} \bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj}$$

where $0 \leq l \leq 1$, $t > 0$, and $l + t \geq 1$. For $1 \leq j \leq t$, q_{fj} is a general MSVS program; whereas q_{ei} $(i = 1)$ and q_{cj} $(1 \leq j \leq t)$ are $true$ or all are state formulas of the form:

$$(x_1 = e_1) \wedge ... \wedge (x_l = e_l) \wedge \dot{p_{x_1}} \wedge ... \wedge \dot{p_{x_l}}$$

where $e_k \in D (1 \leq k \leq l)$. □

Theorem 1. Any MSVL program q can be rewritten into its normal form.

Proof: The proof for transforming most of the statements in MSVL into normal form can be found in [2,6]. The other statements of MSVL can be transformed in a similar way. □

Modeling a system with an MSVL program (formula in PTL) p, according to the normal form, we can construct a graph, namely normal form graph (NFG), which explicitly illustrates the state space of the system. Actually, the NFG also presents the models satisfying formula p [5]. For an MSVL program p, the NFG of p is a directed graph, $G = (CL(p), EL(p))$, where $CL(p)$ denotes the set of nodes and $EL(p)$ denotes the set of edges in the graph. In $CL(p)$, each node is specified by a program in MSVL, while in $EL(p)$, each edge is a directed arc labeled with a state formula p_e from node q to node r and identified by a triple, (q, p_e, r). $CL(p)$ and $EL(p)$ of G can be inductively defined as in Definition 2.

Definition 2. For a program 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. $p \in CL(p)$;
2. For all $q \in CL(p) \setminus \{\varepsilon, false\}$, if $q \equiv \bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj}$, then $\varepsilon \in CL(p)$, $(q, q_{ei}, \varepsilon) \in EL(p)$ for each i; $q_{fj} \in CL(p)$, $(q, q_{cj}, q_{fj}) \in EL(p)$ for all j;

The NFG of formula p is the directed graph $G = (CL(p), EL(p))$. □

Definition 2 implies an algorithm for constructing NFGs of MSVL programs. In the NFG of a program p generated by Definition 2, the set $CL(p)$ of nodes and the set $EL(p)$ of edges are inductively produced by repeatedly rewriting the new created nodes into their normal forms. So one question we have to answer is whether or not the rewriting process terminates. Fortunately, we can prove that, for any MSVL program p, the number of nodes in $CL(p)$ is finite. An outline of the proof is given in Appendix C.

To precisely characterize the models satisfying the program (formula), that is, the behaviors of the system, a finite label F needs further to be added in the NFG as analyzed in [5].

Example 1. NFG of MSVL program $frame(x) \wedge (x = 2 \vee x = 3) \wedge if\ (x = 2)\ then\ len(2)\ else\ \{len(3)\}$ can be constructed as shown in Fig.1. □

4 Property Specification Language

Propositional PTL (PPTL) is employed as the property specification language in our model checking approach.

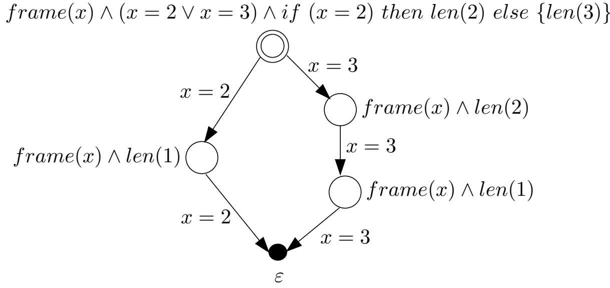

Fig. 1. NFG of MSVL program $frame(x) \wedge (x = 2 \vee x = 3) \wedge if\ (x = 2)\ then\ len(2)\ else\ \{len(3)\}$

4.1 Propositional Projection Temporal Logic

Let *Prop* be a countable set of atomic propositions. The formula p of PPTL is given by the following grammar:

$$p \ ::= \ \pi \mid \bigcirc p \mid \neg p \mid p_1 \vee p_2 \mid (p_1\ ,...,\ p_m)\ prj\ p \mid p^+$$

where $\pi \in Prop$, $p_1\ ,...,\ p_m$ are all well-formed PPTL formulas. A formula is called a state formula if it contains no temporal operators.

Following the definition of Kripke structure [1], we define a state s over *Prop* to be a mapping from *Prop* to $B = \{true, false\}$, $s : Prop \longrightarrow B$. We will use $s[\pi]$ to denote the valuation of π at state s. Intervals, interpretation and satisfaction relation can be defined in the same way as in the first order case.

4.2 Normal Form and NFGs of PPTL Formulas

Definition 3. A PPTL formula q is in normal form if

$$q \stackrel{\text{def}}{=} \bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj}$$

where $0 \leq l \leq 1$, $t > 0$, and $l + t \geq 1$, q_{ei} $(i = 1)$ and q_{cj} $(1 \leq j \leq t)$ are *true* or state formulas of the form:

$$\dot{\pi_1} \wedge ... \wedge \dot{\pi_m}$$

where each $\dot{\pi_k} \in Prop$ $(1 \leq k \leq m)$ and $\dot{\pi_k}$ denotes π_k or $\neg\pi_k$. Each q_{fj} is a general PPTL formula. □

Definition 4. In a normal form, if $\bigvee_{j=1}^{t} q_{cj} \equiv true$ and $\bigvee_{i \neq j} (q_{ci} \wedge q_{cj}) \equiv false$, then this normal form is called a complete normal form. □

The complete normal form plays an important role in transforming the negation of a PPTL formula into its normal form. For example, if q has been written to its complete normal form:

$$q \stackrel{\text{def}}{=} \bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj}$$

then we have,

$$\neg q \stackrel{\text{def}}{=} \bigvee_{i=1}^{l} \neg q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc \neg q_{fj}$$

Theorem 2. Any PPTL formula q can be rewritten into its normal form.
Proof: The proof can be found in [5]. □

A property of a system can be specified by a PPTL formula p. According to the normal form, we can also construct the NFG of p, which explicitly illustrates the models of the formula. The definition for NFGs of PPTL formulas is the same as one defined for MSVL programs.

Theorem 3. For any PPTL formula p, $CL(p)$, the set of nodes in the NFG of p is finite.
Proof: The proof of the theorem can be found in [5]. □

To precisely characterize the models of PPTL formulas, finite labels F are added in the NFGs to confine the finitely often occurrences of some nodes in paths of an NFG as analyzed in [5].

Example 2. NFG of formula $\neg(true; \neg \bigcirc q) \vee p \wedge \bigcirc q$ is shown in Fig.2. □

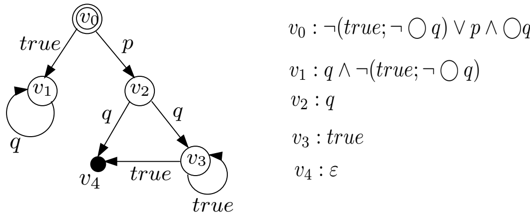

Fig. 2. NFG of formula $\neg(true; \neg \bigcirc q) \vee p \wedge \bigcirc q$

5 Model Checking Approach with PTL Based on NFGs

5.1 Basic Approach

Modeling the system to be verified by an MSVL program p, and specifying the desirable property of the system by a PPTL formula ϕ, to check whether or not the system satisfies the property, we need to prove the validation of

$$p \rightarrow \phi$$

If $p \to \phi$ valid, the system satisfies the property, otherwise the system violates the property. Equivalently, we can check the satisfiability of

$$\neg(p \to \phi) \equiv p \wedge \neg\phi$$

If $p \wedge \neg\phi$ is unsatisfiable ($p \to \phi$ is valid), the system satisfies the property, otherwise the system fails to satisfy the property, and for each $\sigma \models p \wedge \neg\phi$, σ determines a counterexample that the system violates the property. Accordingly, our model checking approach can be translated to the satisfiability of PTL formulas of the form $p \wedge \neg\phi$, where p is an MSVL program and ϕ is a formula in PPTL. Since both model p and property ϕ are formulas in PTL, we call this model checking a unified approach.

To check the satisfiability of PTL formula $p \wedge \neg\phi$, we construct the NFG of $p \wedge \neg\phi$. As depicted in Fig.3, initially, we create the root node $p \wedge \neg\phi$, then

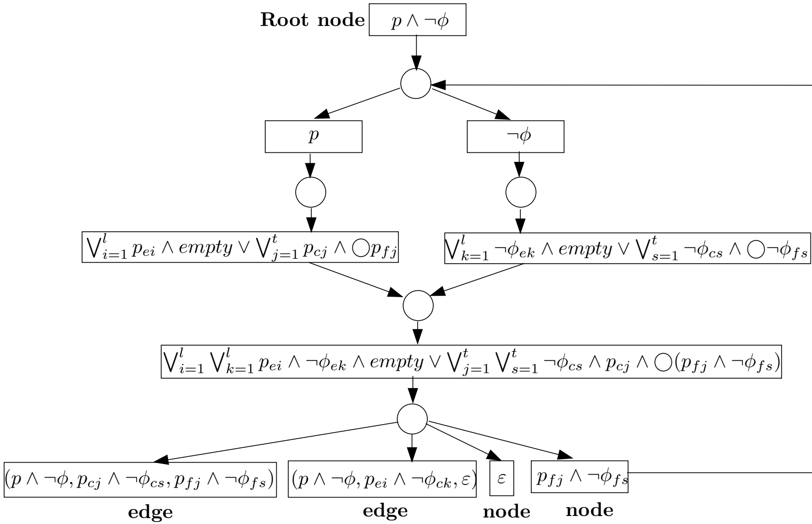

Fig. 3. Constructing NFG of $p \wedge \neg\phi$

we rewrite p and $\neg\phi$ into their normal forms respectively. By computing the conjunction of normal forms of p and $\neg\phi$, new nodes ε and $p_{fj} \wedge \neg\phi_{fs}$, and edges $(p \wedge \neg\phi, p_{ei} \wedge \neg\phi_{ck}, \varepsilon)$ from node $p \wedge \neg\phi$ to ε, $(p \wedge \neg\phi, p_{cj} \wedge \neg\phi_{cs}, p_{fj} \wedge \neg\phi_{fs})$ from $p \wedge \neg\phi$ to $p_{fj} \wedge \neg\phi_{fs}$ are created. Further, by dealing with each new created nodes $p_{fj} \wedge \neg\phi_{fs}$ using the same methods as the root nodes $p \wedge \neg\phi$ repeatedly, the NFG of $p \wedge \neg\phi$ can be produced. Thus, it is apparent that each node in the NFG of $p \wedge \neg\phi$ is in the form of $p' \wedge \neg\phi'$, where p' and ϕ' are nodes in the NFGs of p and $\neg\phi$ respectively. Therefore, a recursive algorithm can be formalized in Pseudo code as shown in algorithm *NFG*. In the algorithm, another function $Nf(p)$ is called to produce the normal form of a PPTL formula or an MSVL program p. This function can be found in [5]. For the complexity of the algorithm, roughly speaking, if $|cl(p)| = O(n)$ and $|cl(\neg\phi)| = O(m)$, at most, $|cl(p \wedge \neg\phi)| = O(n \times m)$.

```
Function NFG(p ∧ ¬φ)
/* precondition: p is a program in MSVL, ¬φ is a formula in PPTL*/
/* postcondition: NFG(p ∧ ¬φ) computes NFG of p ∧ ¬φ, G = (CL(p ∧ ¬φ), EL(p ∧ ¬φ))*/
begin function
  CL(p ∧ ¬φ) = {p ∧ ¬φ}; EL(p ∧ ¬φ) = ∅; mark[p ∧ ¬φ] = 0; AddE = AddN = 0;
  while there exists r ∧ ¬ϕ ∈ CL(p ∧ ¬φ) \ {ε}, and mark[r ∧ ¬ϕ] ==0
   do mark[r ∧ ¬ϕ] =1;                                  /*marking r ∧ ¬ϕ is decomposed*/
    Q =NF(r)∧NF(¬ϕ);
    case
     Q is ⋁_{j=1}^{h} ⋁_{i=1}^{t} r_ej ∧ ¬ϕ_ei ∧ empty: AddE=1;   /*first part of NF needs added*/
     Q is ⋁_{k=1}^{t} ⋁_{l=1}^{n} r_ck ∧ ¬ϕ_cl ∧ ◯(r_fk ∧ ¬ϕ_fl) : AddN=1; /*second part of NF needs added*/
     Q is ⋁_{j=1}^{h} ⋁_{i=1}^{t} r_ej ∧ ¬ϕ_ei ∧ empty∨ ⋁_{k=1}^{t} ⋁_{l=1}^{n} r_ck ∧ ¬ϕ_cl ∧ ◯(r_fk ∧ ¬ϕ_fl): AddE=AddN=1;
                                                        /*both parts of NF needs added*/
    end case
    if AddE == 1 then                                   /*add first part of NF*/
              CL(p ∧ ¬φ) = CL(p ∧ ¬φ) ∪ {ε};
              EL(p ∧ ¬φ) = EL(p ∧ ¬φ) ∪ ⋃_{j=1}^{h} ⋃_{i=1}^{t} {(r ∧ ¬ϕ, r_ej ∧ ¬ϕ_ei, ε)};
              AddE=0;
    if AddN == 1 then                                   /*add second part of NF*/
         for each r_fk ∧ ¬ϕ_fl if r_fk ∧ ¬ϕ_fl ∉ CL(p ∧ ¬φ)
           mark[r_fk ∧ ¬ϕ_fl]=0;                        /*r_fk ∧ ¬ϕ_fl needs decomposed*/
           CL(p ∧ ¬φ) = CL(p ∧ ¬φ) ∪ ⋃_{k=1}^{t} ⋃_{l=1}^{n} {r_fk ∧ ¬ϕ_fl};
           EL(p ∧ ¬φ) = EL(p ∧ ¬φ) ∪ ⋃_{k=1}^{t} ⋃_{l=1}^{n} {(r ∧ ¬ϕ, r_ck ∧ ¬ϕ_cl, r_fk ∧ ¬ϕ_fl)};
         AddN=0;
  end while
  return G;
end function
```

Further, for any node in the NFG of $p \wedge \neg\phi$, finite label F is added in node $p' \wedge \neg\phi'$ where if in the NFG of p, p' is labeled with F, or in the NFG of $\neg\phi$, $\neg\phi'$ is labeled with F. In the NFG of formula $q \equiv p \wedge \neg\phi$, a finite path, $\Pi = \langle q, q_e, q_1, q_{1e}, ..., \varepsilon\rangle$, is an alternate sequence of nodes and edges from the root to ε node, while an infinite path, $\Pi = \langle q, q_e, q_1, q_{1e}, ...\rangle$, is an infinite alternate sequence of nodes and edges emanating from the root, where F occurs only finitely often. Similar to the proof in [5], it can be proved that, the paths in the NFG of q precisely characterize models of q. Thus, if there exist paths in the NFG of q, q is satisfiable, otherwise unsatisfiable.

5.2 Model Checker

We have developed a model checking tool (prototype) based on our model checking algorithm. Generally, the prototype can work in three modes: modeling, simulation and verification. With the modeling mode, given the MSVL program p of a system, the state space of the system can implicitly be given as an NFG of p. In the simulation mode, an execution path of the NFG of the system is output according to minimal model semantics of MSVL [6]. Under the verification mode, given a system model described by an MSVL program, and a property specified by a PPTL formula, it can automatically be checked whether the system

satisfies the property or not, and the counterexample can be given if the system does not satisfy the property.

5.3 Example

As an example, consider the mutual exclusion problem of two processes competing for a shared resource as analyzed in [20]. Pseudo code for this example can be given as shown in Fig.4. We assume that the processes are executed in one time unit in

```
          process A                              process B
              forever                                forever
A.pc = 0          wait for B.pc = 0      B.pc = 0        wait for A.pc = 0
A.pc = 1          access shared resource B.pc = 1        access shared resource
              end forever                            end forever
          end process                            end process
```

Fig. 4. Pseudo code for two processes A and B competing for a shared resource

an interleaving manner. The wait statement makes a process into sleep. When all processes are asleep the scheduler tries to find a process satisfying waiting condition and reactivates the corresponding process. If all of the waiting conditions are false the system stalls. This mutual exclusion problem can be coded in MSVL as follows. Notice that the underlined code can be ignored with the current part since it is for the purpose of making a counterexample later on.

```
frame(Apc, Bpc, Ars, Brs) and
(Apc=0 and Ars=0 and Bpc=0 and Brs=0 and skip;
while(true){
(await(Bpc=0);
Apc=1 and Ars=1 and skip;
Apc=0 and Ars=0 and skip)
or
(Apc=1 and Ars=1 and Bpc=1 and Brs=1 and skip;
Apc=0 and Ars=0 and Bpc=0 and Brs=0 and skip)
or
(await(Apc=0);
Bpc=1 and Brs=1 and skip;
Bpc=0 and Brs=0 and skip) }).
```

where *Ars=1* (*Brs=1*) means processes A (B) is in the shared resource, while *Ars=0* (*Brs=0*) means processes A (B) has released the shared resource. With the modeling mode of MSVL, the state space of the mutual exclusion problem can be created and presented as an NFG as shown in Fig.5. In the NFG, edge 0 indicates that neither process A nor B is in the shared resource; edge 1 (from 1 to 2) indicates that process A is in the shared resource and B is not; edge 2 (from 2 to 1) indicates that neither process A nor B is in the shared resource; edge 4 (from 1 to 3) indicates that process B is in the shared resource and A is not; edge 5 (from 3 to 1) indicates that neither process A nor B is in the shared resource.

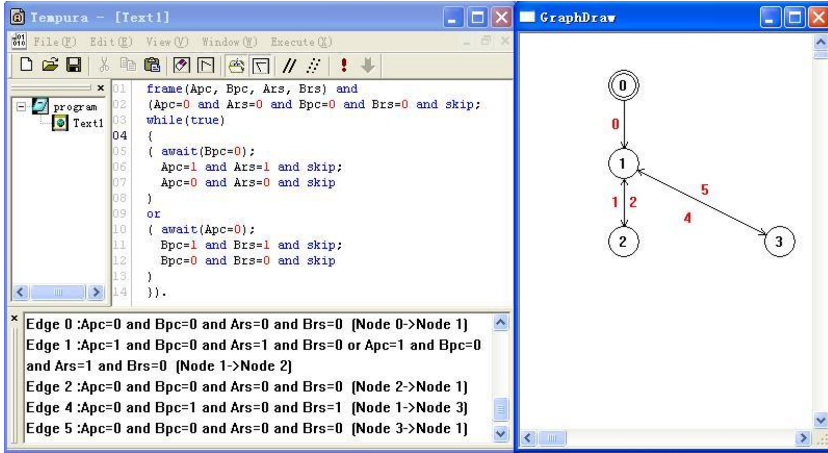

Fig. 5. NFG of the mutual exclusion problem

As a result, the property, "processes A and B will never be in the shared resource in the same time", should hold. That is, Ars and Brs will never be assigned with 1 at the same time. By employing propositions p and q to denote $Ars = 1$ and $Brs = 1$ respectively, this property can be specified by $\Box(\neg(p \wedge q))$ in PPTL. With the verification mode of MSVL, we add the following code

</define p:Ars=1; define q:Brs=1; always(∼(p and q))/>

to the beginning of the MSVL code for the mutual exclusion problem, then run the code with model checker, an empty NFG with no edges is produced as shown in Fig.6. This means that the formula is unsatisfiable, and the system satisfies the property.

Fig. 6. Verification result

Suppose that, when A is in the shared resource, B is possible in the shared resource. To model it, we add the code with underline to the previous MSVL code of the system. Now we check whether or not the system satisfies $\Box(\neg(p \wedge q))$, and the resulting NFG is produced as shown in Fig.7. Obviously, there exist infinite paths where node 1 and node 2 appear infinitely often. Thus, the property cannot be satisfied.

As you can see, MSVL and PPTL can be used to verify properties of programs in a similar way as kripke structures (or automata) and PLTL (or CTL) do. However, the expressive power of PLTL and CTL is limited. They cannot express regular properties such as "a property Q holds at even states ignoring odd ones

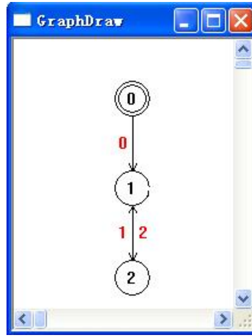

Fig. 7. Verification result

over an interval (or computation run) " [7]. This type of property can be specified and verified by PPTL. In the following, we further verify such a property of the mutual exclusion problem.

The mutual exclusion problem has a special property: "neither A nor B is in the shared resource, immediately after A or B released the shared resource; and when A or B is in the shared resource, it releases the resource at the next state". Basically, this property is a regular property. It can be specified by, $\neg p \wedge \neg q \wedge (\bigcirc^2 \neg p \wedge \neg q \wedge empty)^*$ in PPTL. Thus, we can add *</define p:Ars=1; define q:Brs=1; ~p and ~q and (next(next(~p and ~q and empty)))# />* to the beginning of the MSVL code for the mutual exclusion problem, then run the code with model checker, an empty NFG with no edges is produced as shown in Fig.8. Hence, the formula is unsatisfiable, and the system satisfies the property.

Fig. 8. Verification result

6 Conclusion

In this paper, we proposed a unified model checking approach with PTL based on NFGs. A model checker has also been developed to support the proposed method. This approach has an apparent advantage: model p and property ϕ of a system are both described in the same logic framework PTL. This enables us to translate the problem of checking whether or not p satisfies ϕ to the problem of checking the satisfiability of $p \wedge \neg\phi$. In turn, this can be done by constructing the NFG of $p \wedge \neg\phi$ and checking whether or not there exist any finite or infinite paths in the NFG. As you can see, NFG is a finite graph based structure. So we can use graph theory to manipulate NFGs. Further, an NFG can be equivalently transformed to a Büchi automaton [22]. Hence, automata theory can also be used to manipulate NFGs. However, our approach, in worst case, does not reduce the

complexity of the model checking problems although in many cases it works well since we do not need to produce a whole NFG but just a finite or infinite path as a counterexample.

To combat the space explosion problem, we will further investigate the possibility of combinations of SMC or BMC techniques with our approach. In particular, BMC is a SAT based approach for searching a counterexample in a given integer k steps. With this approach, the model M in the Kripke structure and property ϕ in a PLTL formula of a system are translated to a propositional classic logic formula f. To check whether or not $M \models \phi$ is equivalently to check the satisfiability of f. Thus, the SAT procedure can be used to solve the problem. This idea can be used in our approach. However, we do not need to translate the formulas into a classic propositional logic framework rather in their normal forms and use SAT procedures in a stepwise way. This research is a challenge to us in the near future. Also, the current version of the model checker is merely a prototype and lots of efforts are needed to improve it. In addition, to examine our method, several case studies with larger examples are also required.

Acknowledgement

We would like to thank Miss Xiao Xiao Yang, Miss Xia Guo and Miss Xiao Xing Zhang for the useful discussion. In particular, Guo and Zhang's effort to make the verification example work with the prototype is very appreciated.

References

1. Kripke, S.A.: Semantical analysis of modal logic I: normal propositional calculi. Z. Math. Logik Grund. Math. 9, 67–96 (1963)
2. Duan, Z.: An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming. PhD thesis, University of Newcastle Upon Tyne (May 1996)
3. Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, Beijing (2006)
4. Moszkowski, B.: Reasoning about digital circuits. Ph.D Thesis, Department of Computer Science, Stanford University. TRSTAN-CS-83-970 (1983)
5. Duan, Z., Tian, C., Zhang, L.: A Decision Procedure for Propositional Projection Temporal Logic with Infinite Models. Acta Informatica 45(1), 43–78 (2008)
6. Duan, Z., Yang, X., Koutny, M.: Framed Temporal Logic Programming. Science of Computer Programming 70, 31–61 (2008)
7. Holzmann, G.J.: The Model Checker Spin. IEEE Trans. on Software Engineering 23(5), 279–295 (1997)
8. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation 98(2), 142–170 (1992)
9. Coudert, O., Madre, J.C.: A unified framework for the formal verification of sequential circuits. In: Proc. IEEE International Conference on Computer-Aided Design (1990)

10. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579. Springer, Heidelberg (1999)
11. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
12. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. ASI, vol. F 13, pp. 123–144. Springer, Berlin (1985)
13. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
14. Godefroid, P., Wolper, P.: A partial approach to model checking. Information and Computation 110(2), 305–326 (1994)
15. Esparza, J.: Model checking using net unfoldings. Science of Computer Programming 23, 151–195 (1994)
16. Penczek, W., Gerth, R., Kuiper, R.: Partial order reductions preserving simulations (submitted for publication, 1999)
17. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems 16(3), 843–871 (1994)
18. Josko, B.: Verifying the correctness of AADL modules using model checking. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1989. LNCS, vol. 430, pp. 386–400. Springer, Heidelberg (1990)
19. Josko, B.: Modular Specification and Verification of Reactive Systems. PhD thesis, Univ. Oldenburg, Fachbereich Informatik (April 1993)
20. Biere, A., Cimati, A., Clark, E.M., Strichman, O., Zhu, Y.: Bounded Model Checking. Advances in Computers 58 (2003)
21. Bryant, R.E.: Graph-based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers C35(12), 1035–1044 (1986)
22. Tian, C., Duan, Z.: Propositional Projection Temporal Logic. In: Agrawal, M., Du, D., Duan, Z., Li, A. (eds.) TAMC 2008. LNCS, vol. 4978, pp. 47–58. Springer, Heidelberg (2008)
23. Liu, S., Wang, H.: An automated approach to specification animation for validation. Journal of Systems and Software 80, 1271–1285 (2007)
24. Liu, S., Chen, Y.: A relation-based method combining functional and structural testing for test case generation. Journal of Systems and Software 81, 234–248 (2008)
25. Duan, Z., Koutny, M.: A framed temporal logic programming language. Journal of Computer Science and Technology 19, 333–344 (2004)
26. Gabbay, D., Pnueli, A., Shelah, S., Stavi, J.: On the temporal analysis of fairness. In: POPL 1980: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 163–173. ACM Press, New York (1980)
27. McNaughton, R., Papert, S.A.: Counter-Free Automata (M.I.T research monograph no.65). The MIT Press, Cambridge (1971)

Appendix A: Logic Laws of PTL

The following are some useful logic laws of PTL.

L_1	$\Box(P \wedge Q) \equiv \Box P \wedge \Box Q$
L_2	$\Diamond(P \vee Q) \equiv \Diamond P \vee \Diamond Q$
L_3	$\bigcirc(P \vee Q) \equiv \bigcirc P \vee \bigcirc Q$
L_4	$\bigcirc(P \wedge Q) \equiv \bigcirc P \wedge \bigcirc Q$
L_5	$R;(P \vee Q) \equiv (R;P) \vee (R;Q)$
L_6	$(P \vee Q);R \equiv (P;R) \vee (Q;R)$
L_7	$\Diamond P \equiv P \vee \bigcirc\Diamond P$
L_8	$\Box P \equiv P \wedge \odot \Box P$
L_9	$more \wedge \neg \bigcirc P \equiv more \wedge \bigcirc\neg P$
L_{10}	$\neg \odot P \equiv \bigcirc\neg P$
L_{11}	$\bigcirc(\exists x : p) \equiv \exists x : \bigcirc p$
L_{12}	$\bigcirc P;Q \equiv \bigcirc(P\ ;Q)$
L_{13}	$w \wedge (P\ ;Q) \equiv (w \wedge P)\ ;Q$
L_{14}	$p^+ \equiv p \vee (p;p^+)$
L_{15}	$Q\ prj\ empty \equiv Q$
L_{16}	$empty\ prj\ Q \equiv Q$
L_{17}	$(P_1, ...P_m)\ prj\ empty \equiv P_1; ...; P_m$
L_{18}	$(P, empty)\ prj\ Q \equiv (P \wedge \Diamond empty)\ prj\ Q$
L_{19}	$(P_1, ..., P_t, w \wedge empty, P_{t+1}, ..., P_m)\ prj\ Q \equiv (P_1, ..., P_t, w \wedge P_{t+1}, ..., P_m) prjQ$
L_{20}	$(P_1, ..., (P_i \vee P_i'), ..., P_m)\ prj\ Q \equiv ((P_1, ..., P_i, ..., P_m)\ prj\ Q) \vee ((P_1, ..., P_i', ..., P_m)\ prj\ Q)$
L_{21}	$(P_1, ..., P_m)\ prj\ (P \vee Q) \equiv ((P_1, ..., P_m)\ prj\ P) \vee ((P_1, ..., P_m)\ prj\ Q)$
L_{22}	$(P_1, ..., P_m)\ prj\ \bigcirc Q \equiv (P_1 \wedge more; (P_2, ..., P_m)\ prj\ Q) \vee (P_1 \wedge empty; (P_2, ..., P_m)\ prj\ \bigcirc Q)$
L_{23}	$(\bigcirc P_1, ..., P_m)\ prj\ \bigcirc Q \equiv \bigcirc(P_1\ ; (P_2, ..., P_m)\ prj\ Q)$
L_{24}	$(w \wedge P_1, ..., P_m)\ prj\ Q \equiv w \wedge ((P_1, ..., P_m)\ prj\ Q)$
L_{25}	$(P_1, ..., P_m)\ prj\ (w \wedge Q) \equiv w \wedge ((P_1, ..., P_m)\ prj\ Q)$

Appendix B: Logic Laws of MSVL

L_{26}	$while\ b\ do\ p \equiv if\ b\ then\ (p; while\ b\ do\ p)\ else\ empty$
L_{27}	$while\ b\ do\ p \equiv if\ b\ then\ (p \wedge more; while\ b\ do\ p)\ else\ empty$
L_{28}	$while\ b\ do\ p \equiv ((\neg b \wedge empty) \vee (b \wedge p \wedge more; while\ b\ do\ p)) \vee b \wedge p \wedge \Box more$
L_{29}	$while\ b\ do\ p \equiv ((\neg b \wedge empty) \vee (b \wedge p; while\ b\ do\ p)) \vee b \wedge p \wedge \Box more$
L_{30}	$frame(x) \equiv frame(x) \vert\vert frame(x) \equiv frame(x); frame(x) \equiv frame(x) \wedge frame(x)$
L_{31}	$frame(x) \wedge more \equiv \bigcirc(lbf(x) \wedge frame(x))$
L_{32}	$frame(x) \wedge empty \equiv empty$
L_{33}	$frame(x) \wedge (p \vee q) \equiv frame(x) \wedge p \vee frame(x) \wedge q$
L_{34}	$frame(x) \wedge (p; q) \equiv frame(x) \wedge p; frame(x) \wedge q$
L_{35}	$frame(x) \wedge (p \vert\vert q) \equiv frame(x) \wedge p \vert\vert frame(x) \wedge q$

Appendix C: Finiteness of NFGs of MSVL Programs

Let $D = \{d_1, ..., d_n\}$ be a finite set of data, $V = \{x_1, ..., x_m\}$ a finite set of variables, and *Prop* a countable set of atomic propositions. To prove the finiteness of NFGs of MSVL programs, we first prove that, for any MSVL program, it can be equivalently expressed by a PPTL formula.

Theorem 4. Any program p in MSVL can be equivalently expressed by a formula $\Phi(p)$ in PPTL.

Proof. The proof proceeds by induction on structures of programs in MSVL. First of all, we assume that any boolean expression b can be evaluated to a boolean value true or false, and an arithmetic expression e can be evaluated to a value $d_k \in D$. Therefore, a boolean expression b can be thought of as an atomic proposition $p_b \in Prop$. Further,

1. For *empty*, $\Phi(empty) \stackrel{\text{def}}{=} empty$;

2. For $x_i = e$, we define $x_i = d_j \stackrel{\text{def}}{=} p_i^j \in Prop$, where $x_i \in V$ and $d_j \in D$. Thus, $\Phi(x_i = e) \stackrel{\text{def}}{=} p_i^k$ if $e = d_k \in D$ otherwise *false*

3. For $x_i \Leftarrow e$, by the definition, $x_i \Leftarrow e \equiv x_i = e \wedge p_{x_i}$,

$$\Phi(x_i \Leftarrow e) \stackrel{\text{def}}{=} \Phi(x_i = e) \wedge p_{x_i}$$

4. For $lbf(x_i)$, by the definition, $lbf(x_i) \equiv \neg p_{x_i} \rightarrow \exists d_n \in D : (\odot x_i = d_n \wedge x_i = d_n)$, we have,

$$\Phi(lbf(x_i)) \stackrel{\text{def}}{=} \neg p_{x_i} \rightarrow p_i^n$$

5. For $frame(x_i)$, by the definition, $frame(x_i) \equiv \Box(more \rightarrow \bigcirc lbf(x_i))$, we have,

$$\Phi(frame(x_i)) \stackrel{\text{def}}{=} \Box(more \rightarrow \bigcirc\Phi(lbf(x_i)))$$

6. For $p \wedge q$, $\Phi(p \wedge q) \stackrel{\text{def}}{=} \Phi(p) \wedge \Phi(q)$;
7. For $p \vee q$, $\Phi(p \vee q) \stackrel{\text{def}}{=} \Phi(p) \vee \Phi(q)$;
8. For $\bigcirc p$, $\Phi(\bigcirc p) \stackrel{\text{def}}{=} \bigcirc \Phi(p)$;
9. For $\Box p$, $\Phi(\Box p) \stackrel{\text{def}}{=} \Box \Phi(p)$;
10. For $p; q$, $\Phi(p; q) \stackrel{\text{def}}{=} \Phi(p); \Phi(q)$;
11. For *if b then p else q*, by the definition, *if b then p else q* $\equiv b \wedge p \vee \neg b \wedge q$, we have,

$$\Phi(if\ b\ then\ p\ else\ q) \stackrel{\text{def}}{=} p_b \wedge \Phi(p) \vee \neg p_b \wedge \Phi(q)$$

12. For $(p_1, ..., p_m) prj\ q$, $\Phi((p_1, ..., p_m) prj\ q) \stackrel{\text{def}}{=} (\Phi(p_1), ..., \Phi(p_m)) prj\ \Phi(q)$;
13. For *while b do p*, by the definition, *while b do p* $\equiv (p \wedge b)^* \wedge \Box(empty \rightarrow \neg b)$, we have,

$$\Phi(while\ b\ do\ p) \stackrel{\text{def}}{=} (\Phi(p) \wedge P_b)^* \wedge \Box(empty \rightarrow \neg p_b)$$

14. For $p||q$, by the definition, $p||q \equiv p \wedge (q; true) \vee q \wedge (p; true)$, we have,

$$\Phi(p||q) \stackrel{\text{def}}{=} \Phi(p) \wedge (\Phi(q); true) \vee \Phi(q) \wedge (\Phi(p); true)$$

15. For $await(b)$, by the definition, $await(b) \equiv (frame(x_1) \wedge ... \wedge frame(x_h)) \wedge \Box(empty \leftrightarrow b)$,

$$\Phi(await(b)) \stackrel{\text{def}}{=} (\Phi(frame(x_1)) \wedge ... \wedge \Phi(frame(x_h))) \wedge \Box(empty \leftrightarrow p_b)$$

16. For $\exists x : q$, since q can be rewritten into its normal form, $q \stackrel{\text{def}}{=} \bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj}$, we have,

$$\begin{aligned}
\Phi(\exists x : q) &\stackrel{\text{def}}{=} \Phi(\exists x : (\bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj})) \\
&\equiv \Phi(\bigvee_{i=1}^{l} (\exists x : q_{ei}) \wedge empty \vee \bigvee_{j=1}^{t} (\exists x : q_{cj}) \wedge \bigcirc(\exists x : q_{fj})) \\
&\equiv \bigvee_{i=1}^{l} \Phi(\exists x : q_{ei}) \wedge empty \vee \bigvee_{j=1}^{t} \Phi(\exists x : q_{cj}) \wedge \bigcirc \Phi(\exists x : q_{fj}) \\
&\equiv \bigvee_{i=1}^{l} \bigvee_{k=1}^{n} q_{ei}[d_k/x] \wedge empty \vee \bigvee_{j=1}^{t} \bigvee_{k=1}^{n} q_{cj}[d_k/x] \wedge \bigcirc \Phi(\exists x : q_{fj})
\end{aligned}$$

Thus, for any MSVL program, it can be equivalently expressed by a PPTL formula. □

Notice that in the above proof of 16, we can use $\Phi(\exists x_i p)$ recursively so that a PPTL formula can be obtained. A question one may ask is that this transformation process can terminate? The answer is 'yes' since a simple inductive proof on the structure of p can be made to achieve the conclusion. We omit the details here. In [5], we have proved the finiteness of NFGs of PPTL formulas. Hence, the conclusion also holds for MSVL programs since any MSVL program can be equivalently expressed by a PPTL formula.