SAT-Based Verification of Safe Petri Nets

Shougo Ogata, Tatsuhiro Tsuchiya, and Tohru Kikuno

Department of Information Systems Engineering
Graduate School of Information Science and Technology, Osaka University
{s-ogata, t-tutiya, kikuno}@ist.osaka-u.ac.jp

Abstract. Bounded model checking has received recent attention as an efficient verification method. The basic idea behind this new method is to reduce the model checking problem to the propositional satisfiability decision problem or SAT. However, this method has rarely been applied to Petri nets, because the ordinary encoding would yield a large formula due to the concurrent and asynchronous nature of Petri nets. In this paper, we propose a new SAT-based verification method for safe Petri nets. This method can reduce verification time by representing the behavior by very succinct formulas. Through an experiment using a suite of Petri nets, we show the effectiveness of the proposed method.

Keywords: Bounded model checking, SAT, Petri nets

1 Introduction

Model checking [5] is a powerful technique for verifying systems that are modeled as a finite state machine. The main challenge in model checking is to deal with the *state space explosion* problem, because the number of states can be very large for realistic designs. Recently bounded model checking has been receiving attention as a new solution to this problem [2,6]. The main idea is to look for counterexamples (or witnesses) that are shorter than some fixed length k for a given property. If a counterexample can be found, then it is possible to conclude that the property does not hold in the system. The key behind this approach is to reduce the model checking problem to the propositional satisfiability problem. The formula to be checked is constructed by unwinding the transition relation of the system k times such that satisfying assignments represent counterexamples.

In the literature, it has been reported that this method can work efficiently, especially for the verification of digital circuits. An advantage of this method is that it works efficiently even when compact BDD representation cannot be obtained. It is also an advantage that the approach can exploit recent advances in decision procedures of satisfiability.

On the other hand, this method does not work well for asynchronous systems like Petri nets, because the encoding scheme into propositional formulas is not suited for such systems; it would require a large formula to represent the transition relation, thus resulting in large execution time and low scalability.

To address this issue, approaches that use other techniques than SAT decision procedures have been proposed in [8,9]. These approaches allow bounded model checking of Petri nets by using answer set programming [9] and by using Boolean circuit satisfiability checking [8].

In this paper, we tackle the same problem but in a different way; we propose a new verification method using ordinary SAT solvers. As in [8,9], we limit our discussions to verification of 1-bounded or safe Petri nets in this paper. The new method enhances the effectiveness of SAT-based verification in the following two ways.

- Our method uses a much succinct formula, compared to the existing method. This shortens the execution time of a SAT procedure.
- Our method allows the formula to represent counterexamples of length greater than k while guaranteeing to detect counterexamples of length k or less. This enlarges the state space that can be explored.

To demonstrate the effectiveness of the approach, we show the results of applying it to a suite of Petri nets.

The remainder of the paper is organized as follows. Section 2 describes the basic definition of Petri nets and how to represent them symbolically. Section 3 explains our proposed method for reachability checking. Section 4 discusses liveness checking. In Section 5 a pre-processing procedure is proposed. Experimental results are presented in Section 6. Section 7 concludes the paper with a summary and directions for future work.

2 Preliminaries

2.1 Petri Nets

A *Petri net* is a 4-tuple $(\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$ where $\mathcal{P} = \{p_1, p_2, \cdots, p_m\}$ is a finite set of places, $\mathcal{T} = \{t_1, t_2, \cdots, t_n\}$ $(\mathcal{P} \cap \mathcal{T} = \emptyset)$ is a finite set of transitions, $\mathcal{F} \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$ is a set of arcs, and M_0 is the initial state (marking). The set of input places and the set of output places of t are denoted by ${}^\bullet t$ and $t^\bullet$, respectively.

We define a relation $\xrightarrow{t}$ over states as follows: $S \xrightarrow{t} S'$ iff some $t \in \mathcal{T}$ is enabled at S and S' is the next state resulted in by its firing. Also we define a *computation* as a sequence of states $S_0 S_1 \cdots S_l$ such that for any $0 \leq i < l$ either (i) $S_i \xrightarrow{t} S_{i+1}$ for some t, or (ii) $S_i = S_{i+1}$ and no t is enabled at S_i. S_i is reachable from S_0 in i steps iff there is a computation $S_0 S_1 \cdots S_i$. We define the length of a computation $S_0 S_1 \cdots S_i$ as i.

A Petri net is said to be *1-bounded* or *safe* if the number of tokens in each place does not exceed one for any state reachable from the initial state. Note that no source transition (a transition without any input place) exists if a Petri net is safe. For example, the Petri net shown in Figure 1 is safe. Figure 2 shows the reachability graph of this Petri net. In this graph, each state is represented by the places marked with a token.

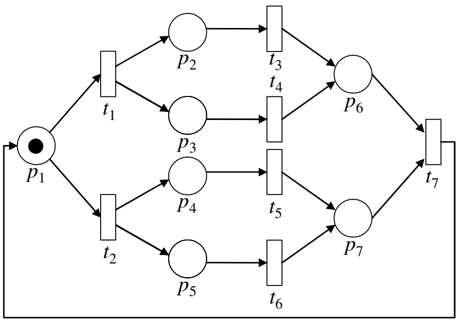

Fig. 1. A Petri net.

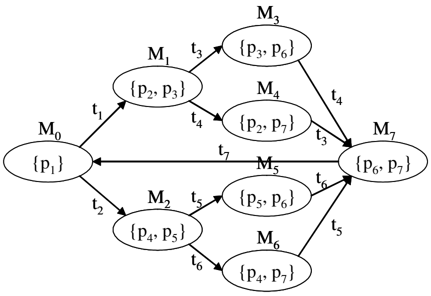

Fig. 2. Reachability graph.

2.2 Symbolic Representation

This subsection describes how a safe Petri net can be represented symbolically. For a safe Petri net, a state S can be viewed as a Boolean vector $S = (s_1, \cdots, s_m)$ of length m such that $s_i = 1$ iff place p_i is marked with a token.

Any set of states can be represented as a Boolean function such that $f(S) = 1$ iff S is in the set. We say that $f(S)$ is a *characteristic function* of the set.

We denote by $E_t(S)$ the characteristic function of the set of states in which transition t is enabled; that is:

$$E_t(S) := \bigwedge_{p_i \in {}^\bullet t} s_i$$

For example, for the Petri net we have:

$$E_{t_1}(S) := s_1,\ E_{t_2}(S) := s_1,\ E_{t_3}(S) := s_2,\ E_{t_4}(S) := s_3,$$
$$E_{t_5}(S) := s_4,\ E_{t_6}(S) := s_5,\ E_{t_7}(S) := s_6 \wedge s_7$$

Any relation over states can be similarly encoded since a relation is simply a set of tuples. Let $T_t(S, S')$ be the characteristic function for the relation $\xrightarrow{t}$.

$T_t(S, S')$ is represented as follows:

$$T_t(S, S') := E_t(S) \wedge \bigwedge_{p_i \in {}^\bullet t \setminus t^\bullet} \neg s'_i \wedge \bigwedge_{p_i \in t^\bullet} s'_i \wedge \bigwedge_{p_i \in \mathcal{P} \setminus ({}^\bullet t \cup t^\bullet)} (s_i \leftrightarrow s'_i)$$

For t_1 in the Petri net in Figure 1, for example, we have:

$$T_{t_1}(S, S') := s_1 \wedge \neg s'_1 \wedge s'_2 \wedge s'_3 \wedge (s_4 \leftrightarrow s'_4) \wedge (s_5 \leftrightarrow s'_5) \wedge (s_6 \leftrightarrow s'_6) \wedge (s_7 \leftrightarrow s'_7)$$

3 Reachability Checking

3.1 Applying the Existing Encoding to Reachability Checking

Let R be the set of states whose reachability to be checked and let $R(S)$ denote its characteristic function. Although there are some variations [16], the basic formula used for checking reachability is as follows:

$$I(S_0) \wedge T(S_0, S_1) \wedge T(S_1, S_2) \wedge \cdots \wedge T(S_{k-1}, S_k) \wedge \big(R(S_0) \vee \cdots \vee R(S_k)\big)$$

where $I(S)$ is the characteristic function of the set of the initial states, and $T(S, S')$ is the *transition relation function* such that

$$T(S, S') = 1 \text{ iff } S' \text{ is reachable from } S' \text{ in one step.}$$

Clearly, $I(S_0) \wedge T(S_0, S_1) \wedge T(S_1, S_2) \cdots \wedge T(S_{k-1}, S_k) = 1$ iff $S_0, S_1, \cdots, S_k$ is a computation from the initial states. Hence the above formula is satisfiable iff some state in R is reachable from one of the initial states in at most k steps. By checking the satisfiability of the formula, therefore, the verification can be carried out.

For Petri nets, M_0 is the only initial state. Thus we have:

$$I(S) := \bigwedge_{p_i \in P_0} s_i \wedge \bigwedge_{p_i \in \mathcal{P} \setminus P_0} \neg s_i$$

where P_0 is the set of places marked with a token in M_0. For example, for the Petri net in Figure 1, $I(S)$ will be:

$$I(S) := s_1 \wedge \neg s_2 \wedge \neg s_3 \wedge \neg s_4 \wedge \neg s_5 \wedge \neg s_6 \wedge \neg s_7$$

$T(S, S') = 1$ iff either (i) $S \xrightarrow{t} S'$ for some t, or (ii) $S = S'$ and no t is enabled at S. Hence we have:

$$\begin{aligned} T(S, S') := {} & T_{t_1}(S, S') \vee \cdots \vee T_{t_n}(S, S') \\ & \vee \Big(\bigwedge_{p_i \in \mathcal{P}} (s_i \leftrightarrow s'_i) \wedge \neg E_{t_1}(S) \wedge \cdots \wedge \neg E_{t_n}(S) \Big) \end{aligned}$$

In practice, this formula would be very large in size. Figure 3 shows the formula $T(S, S')$ obtained from the Petri net in Figure 1. It should be noted that the efficiency of SAT-based verification critically depends on the size of the formula to be checked.

$$
\begin{aligned}
&(s_1 \wedge \neg s'_1 \wedge s'_2 \wedge s'_3 \wedge (s_4 \leftrightarrow s'_4) \wedge (s_5 \leftrightarrow s'_5) \wedge (s_6 \leftrightarrow s'_6) \wedge (s_7 \leftrightarrow s'_7))\\
&\vee(s_1 \wedge \neg s'_1 \wedge s'_4 \wedge s'_5 \wedge (s_2 \leftrightarrow s'_2) \wedge (s_3 \leftrightarrow s'_3) \wedge (s_6 \leftrightarrow s'_6) \wedge (s_7 \leftrightarrow s'_7))\\
&\vee(s_2 \wedge \neg s'_2 \wedge s'_6 \wedge (s_1 \leftrightarrow s'_1) \wedge (s_3 \leftrightarrow s'_3) \wedge (s_4 \leftrightarrow s'_4) \wedge (s_5 \leftrightarrow s'_5) \wedge (s_7 \leftrightarrow s'_7))\\
&\vee(s_3 \wedge \neg s'_3 \wedge s'_7 \wedge (s_1 \leftrightarrow s'_1) \wedge (s_2 \leftrightarrow s'_2) \wedge (s_4 \leftrightarrow s'_4) \wedge (s_5 \leftrightarrow s'_5) \wedge (s_6 \leftrightarrow s'_6))\\
&\vee(s_4 \wedge \neg s'_4 \wedge s'_6 \wedge (s_1 \leftrightarrow s'_1) \wedge (s_2 \leftrightarrow s'_2) \wedge (s_3 \leftrightarrow s'_3) \wedge (s_5 \leftrightarrow s'_5) \wedge (s_7 \leftrightarrow s'_7))\\
&\vee(s_5 \wedge \neg s'_5 \wedge s'_7 \wedge (s_1 \leftrightarrow s'_1) \wedge (s_2 \leftrightarrow s'_2) \wedge (s_3 \leftrightarrow s'_3) \wedge (s_4 \leftrightarrow s'_4) \wedge (s_6 \leftrightarrow s'_6))\\
&\vee(s_6 \wedge s_7 \wedge \neg s'_6 \wedge \neg s'_7 \wedge s'_1 \wedge (s_2 \leftrightarrow s'_2) \wedge (s_3 \leftrightarrow s'_3) \wedge (s_4 \leftrightarrow s'_4) \wedge (s_5 \leftrightarrow s'_5))\\
&\vee((s_1 \leftrightarrow s'_1) \wedge (s_2 \leftrightarrow s'_2) \wedge (s_3 \leftrightarrow s'_3) \wedge (s_4 \leftrightarrow s'_4) \wedge (s_5 \leftrightarrow s'_5) \wedge (s_6 \leftrightarrow s'_6) \wedge (s_7 \leftrightarrow s'_7)\\
&\quad \wedge\neg s_1 \wedge \neg s_2 \wedge \neg s_3 \wedge \neg s_4 \wedge \neg s_5 \wedge \neg(s_6 \wedge s_7))
\end{aligned}
$$

Fig. 3. $T(S, S')$

3.2 Proposed Encoding

We define $d_t(S, S')$ as follows:

$$
\begin{aligned}
d_t(S,S') &:= T_t(S,S') \vee \bigwedge_{p_i \in \mathcal{P}} (s_i \leftrightarrow s'_i)\\
&:= \Big(\big(\bigwedge_{p_i \in {}^\bullet t} s_i \wedge \bigwedge_{p_i \in {}^\bullet t \setminus t^\bullet} \neg s'_i \wedge \bigwedge_{p_i \in t^\bullet} s'_i\big) \vee \bigwedge_{p_i \in {}^\bullet t \cup t^\bullet} (s_i \leftrightarrow s'_i)\Big)\\
&\qquad \wedge \bigwedge_{p_i \in \mathcal{P} \setminus ({}^\bullet t \cup t^\bullet)} (s_i \leftrightarrow s'_i)
\end{aligned}
$$

For the Petri net shown in Figure 1, for example, we have:

$$
\begin{aligned}
d_{t_1}(S,S') := \Big(&(s_1 \wedge \neg s'_1 \wedge s'_2 \wedge s'_3) \vee \big((s_1 \leftrightarrow s'_1) \wedge (s_2 \leftrightarrow s'_2) \wedge (s_3 \leftrightarrow s'_3)\big)\Big)\\
&\wedge(s_4 \leftrightarrow s'_4) \wedge (s_5 \leftrightarrow s'_5) \wedge (s_6 \leftrightarrow s'_6) \wedge (s_7 \leftrightarrow s'_7)
\end{aligned}
$$

It is easy to see that $d_t(S, S') = 1$ iff $S \xrightarrow{t} S'$ or $S = S'$. In other words, $d_t(S, S')$ differs from $T_t(S, S')$ only in that $d_t(S, S')$ evaluates to true also when $S = S'$. We now define a relation $\rightharpoonup$ over states as follows: $S \rightharpoonup S'$ iff $S \xrightarrow{t} S'$ for some t or $S = S'$. Clearly $d_t(S, S') = 1$ iff $S \rightharpoonup S'$.

A step (or more) can be represented by a conjunction of $d_t(S, S')$. Note that this is in contrast to the ordinary encoding where a disjunction of $T_t(S, S')$ is used to represent one step. Specifically, our proposed scheme uses the following formula φ_k:

$$\varphi_k := \mathcal{M}_k \wedge R(S_{k*n})$$

where

$$
\begin{aligned}
\mathcal{M}_k := I(S_0)&\\
&\wedge d_{t_1}(S_0, S_1) \wedge d_{t_2}(S_1, S_2) \wedge \cdots \wedge d_{t_n}(S_{n-1}, S_n)\\
&\wedge d_{t_1}(S_n, S_{n+1}) \wedge d_{t_2}(S_{n+1}, S_{n+2}) \wedge \cdots \wedge d_{t_n}(S_{2n-1}, S_{2n})\\
&\cdots\\
&\wedge d_{t_1}(S_{(k-1)*n}, S_{(k-1)*n+1}) \wedge \cdots \wedge d_{t_n}(S_{k*n-1}, S_{k*n})
\end{aligned}
$$

If φ_k is satisfiable, then there exists a state in R that can be reached in at most $k * n$ steps from the initial state M_0, because φ_k evaluates to true iff (i) $S_0 = M_0$, (ii) for any $0 \leq i < k * n$, $S_i \rightharpoonup S_{i+1}$, and (iii) $S_{k*n} \in R$.

On the other hand, if φ_k is unsatisfiable, then one can conclude that no state in R can be reached in k steps or less. This can be explained as follows. Suppose that a computation $M_0 M_1 \cdots M_l (0 \leq l \leq k)$ exists that starts from the initial state M_0 to a state M_l in R. Let t_{i_j} be the transition such that $M_j \overset{t_{i_j}}{\rightarrow} M_{j+1}$. Then, φ_k is satisfied by the following assignment: for $0 \leq j < l$, $S_{j*n}, S_{j*n+1}, \cdots, S_{j*n+i_j-1} = M_j$, $S_{j*n+i_j}, \cdots, S_{(j+1)*n} = M_{j+1}$; and for $l \leq j < k$, $S_{j*n}, S_{j*n+1}, \cdots, S_{(j+1)*n} = M_l$.

An important observation is that the method may be able to find witness computations of length greater than k. The length of witnesses that can be found is at most $k * n$, and this upper bound is achievable.

Another important advantage of our method is that it is possible to construct a very succinct formula that has the same satisfiability of φ_k. Let $S_i = (s_{1,i}, s_{2,i}, \cdots, s_{m,i})$. For each $d_t(S_j, S_{j+1})$ in φ_k, term $(s_{i,j} \leftrightarrow s_{i,j+1})$ can be removed for all $p_i \in \mathcal{P} \backslash ({}^\bullet t \cup t^\bullet)$ by quantifying $s_{i,j+1}$. The reason for this is as follows: Let $\hat{\varphi}_k$ be the subformula of φ_k that is obtained by removing the term $(s_{i,j} \leftrightarrow s_{i,j+1})$; that is, $\varphi_k = \hat{\varphi}_k \wedge (s_{i,j} \leftrightarrow s_{i,j+1})$. Because this term $(s_{i,j} \leftrightarrow s_{i,j+1})$ occurs as a conjunct in φ_k, φ_k evaluates to true only if $s_{i,j}$ and $s_{i,j+1}$ have the same value. Hence $\hat{\varphi}_k$ with $s_{i,j+1}$ being replaced with $s_{i,j}$ has the same satisfiability as φ_k. The other terms remaining in $\hat{\varphi}_k$ can also be removed in the same way.

Below is the formula that is thus obtained from the Petri net in Figure 1 when $k = 1$.

$$
\begin{aligned}
& I(S_0) \\
& \wedge \Big((s_{1,0} \wedge \neg s_{1,1} \wedge s_{2,1} \wedge s_{3,1}) \\
& \qquad \vee \big((s_{1,0} \leftrightarrow s_{1,1}) \wedge (s_{2,0} \leftrightarrow s_{2,1}) \wedge (s_{3,0} \leftrightarrow s_{3,1}) \big) \Big) \\
& \wedge \Big((s_{1,1} \wedge \neg s_{1,2} \wedge s_{4,2} \wedge s_{5,2}) \\
& \qquad \vee \big((s_{1,1} \leftrightarrow s_{1,2}) \wedge (s_{4,0} \leftrightarrow s_{4,2}) \wedge (s_{5,0} \leftrightarrow s_{5,2}) \big) \Big) \\
& \wedge \Big((s_{2,1} \wedge \neg s_{2,3} \wedge s_{6,3}) \vee \big((s_{2,1} \leftrightarrow s_{2,3}) \wedge (s_{6,0} \leftrightarrow s_{6,3}) \big) \Big) \\
& \wedge \Big((s_{3,1} \wedge \neg s_{3,4} \wedge s_{7,4}) \vee \big((s_{3,1} \leftrightarrow s_{3,4}) \wedge (s_{7,0} \leftrightarrow s_{7,4}) \big) \Big) \\
& \wedge \Big((s_{4,2} \wedge \neg s_{4,5} \wedge s_{6,5}) \vee \big((s_{4,2} \leftrightarrow s_{4,5}) \wedge (s_{6,3} \leftrightarrow s_{6,5}) \big) \Big) \\
& \wedge \Big((s_{5,2} \wedge \neg s_{5,6} \wedge s_{7,6}) \vee \big((s_{5,2} \leftrightarrow s_{5,6}) \wedge (s_{7,4} \leftrightarrow s_{7,6}) \big) \Big) \\
& \wedge \Big((s_{6,5} \wedge s_{7,6} \wedge \neg s_{6,7} \wedge \neg s_{7,7} \wedge s_{1,7}) \\
& \qquad \vee \big((s_{1,2} \leftrightarrow s_{1,7}) \wedge (s_{6,5} \leftrightarrow s_{6,7}) \wedge (s_{7,6} \leftrightarrow s_{7,7}) \big) \Big) \\
& \wedge R(S_7)_{[s_{2,7} \leftarrow s_{2,3}, s_{3,7} \leftarrow s_{3,4}, s_{4,7} \leftarrow s_{4,5}, s_{5,7} \leftarrow s_{5,6}]}
\end{aligned}
$$

where $F_{[x \leftarrow y]}$ represents a formula generated by substituting y for x in F.

Note that this resulting formula is quite smaller than $T(S, S')$ in Figure 3, since $T_t(S, S')$ contains at least $m(= |\mathcal{P}|)$ literals, while its counterpart in our encoding only contains at most $4|^\bullet t|+3|t^\bullet|$ literals. The difference becomes larger if the number of places that are neither input nor output places increases for each transition.

It should also be noted that in this particular case, $k = 1$ is enough to explore all the reachable states. That is, for any reachable state M, a sequence of states $S_0S_1 \cdots S_7$ exists such that: $S_0 = M_0$; for any i, $S_{i-1} \xrightarrow{t_i} S_i$ or $S_{i-1} = S_i$; and $S_7 = M$.

In the worst case, our encoding introduces $n + 1$ different Boolean variables for a single place in order to encode one step. Compared with the ordinary encoding which requires only two different variables (for the current and next states), this number might seem prohibitively large. In practice, however, this rarely causes a problem, because for many transitions in Petri nets representing practical concurrent systems, input and output places are often only a small fraction of all places, and from our experience, the performance of SAT solvers critically depends on the number of literals, but not on the number of different variables.

3.3 Complexity Issues

Reachability checking is a very difficult problem in terms of computational complexity. For arbitrary Petri nets, the problem is decidable but is in EXPSPACE. Although this problem is easier if the Peri net is safe, the complexity is still PSPACE-complete [3].

The proposed method reduces the reachability problem to the propositional satisfiability problem. It is well known that the latter problem is in NP-complete. In many practical situations, however, our method works well for the following reasons: first, recently developed SAT solvers can often perform very effectively due to powerful heuristics; and second, if the state to be checked is reachable and is close to the initial state, it will suffice to check a small formula to decide the reachability.

3.4 Related Properties

Here we show that various properties can be verified with our method, by accordingly adjusting $R(S)$ which represents the state set whose reachability is to be checked.

L0- and L1-liveness. A transition t is said to be $L1$-*live* iff t can be fired at least once. If t is not $L1$-live, then t is said to be $L0$-*live* or *dead*. $L1$-liveness can be verified by checking the reachability to a state where the transition of interest is enabled. The characteristic function of the states where a transition t is enabled is:

$$\bigwedge_{p_i \in {}^\bullet t} s_i$$

Deadlock. A Petri net has a deadlock if there exists a reachable state where no transition can be fired. The set of states where no transition is enabled can be represented as:

$$\neg \bigvee_{t \in \mathcal{T}} E_t(S)$$

4 L3-Liveness Verification

In this section, we discuss verifying properties that can not be checked by reachability analysis. As an example, we consider the problem of checking *$L3$-liveness*; a transition t is said to be *$L3$-live* iff a computation starting from the initial state exists on which t is fired infinitely often.

In our method, the $L3$-liveness of a given transition is checked by searching a computation that contains a loop where t is fired. This idea is the same as LTL bounded model checking; but in our case a slight modification is required to the method described in the previous section, in order to select only loops that involve the firing of the transition of interest.

Specifically we introduce $d'_t(S, S', f)$ as an alternative representation to $d_t(S, S')$. Here f is a Boolean variable. $d'_t(S, S', f)$ is defined as follows.

$$\begin{aligned} d'_t(S, S', f) := \big(f \leftrightarrow (& \bigwedge_{p_i \in {}^\bullet t} s_i \wedge \bigwedge_{p_i \in {}^\bullet t \setminus t^\bullet} \neg s'_i \wedge \bigwedge_{p_i \in t^\bullet} s'_i)\big) \\ & \wedge \big(f \vee \bigwedge_{p_i \in {}^\bullet t \cup t^\bullet} (s_i \leftrightarrow s'_i)\big) \wedge \bigwedge_{p_i \in \mathcal{P} \setminus ({}^\bullet t \cup t^\bullet)} (s_i \leftrightarrow s'_i) \end{aligned}$$

It is easy to see that $d_t(S, S') = \exists f.[d'_t(S, S', f)]$. Also the following two properties hold:

- $d'_t(S, S', 0) = 1$ iff $S = S'$ holds and $S \xrightarrow{t} S'$ does not hold.
- $d'_t(S, S', 1) = 1$ iff $S \xrightarrow{t} S'$.

Let $\mathcal{M}_k^{t_m}$ be defined as follows:

$$\begin{aligned} \mathcal{M}_k^{t_m} := & I(S_0) \\ & \wedge d_{t_1}(S_0, S_1) \wedge d_{t_2}(S_1, S_2) \wedge \cdots \wedge d_{t_{m-1}}(S_{m-2}, S_{m-1}) \\ & \quad \wedge d'_{t_m}(S_{m-1}, S_m, f_{m-1}) \wedge d_{t_{m+1}}(S_m, S_{m+1}) \wedge \cdots \wedge d_{t_n}(S_{n-1}, S_n) \\ & \wedge d_{t_1}(S_n, S_{n+1}) \wedge d_{t_2}(S_{n+1}, S_{n+2}) \wedge \cdots \wedge d_{t_{m-1}}(S_{n+m-2}, S_{n+m-1}) \\ & \quad \wedge d'_{t_m}(S_{n+m-1}, S_{n+m}, f_{n+m-1}) \wedge d_{t_{m+1}}(S_{n+m}, S_{n+m+1}) \wedge \cdots \\ & \quad \wedge d_{t_n}(S_{2n-1}, S_{2n}) \\ & \cdots \\ & \wedge d_{t_1}(S_{(k-1)*n}, S_{(k-1)*n+1}) \wedge \cdots \\ & \quad \wedge d'_{t_m}(S_{(k-1)*n+m-1}, S_{(k-1)*n+m}, f_{(k-1)*n+m-1}) \wedge \cdots \\ & \quad \wedge d_{t_n}(S_{k*n-1}, S_{k*n}) \end{aligned}$$

$\mathcal{M}_k^{t_m}$ is different from $\mathcal{M}_k$ only in that $d_{t_m}(S, S')$ is replaced with $d'_{t_m}(S, S', f)$. Since $d_t(S, S') = \exists f.[d'_t(S, S', f)]$, if $\mathcal{M}_k^{t_m}$ is satisfied by $S_0, S_1, \cdots, S_{k*n}$, then $S_i \rightharpoonup S_{i+1}$ for any $0 \leq i < k * n$.

Now let us define ${}_i\mathcal{L}_k^{t_m}$ as follows:

$$\begin{aligned}{}_i\mathcal{L}_k^{t_m} := (&S_{i*n} = S_{k*n}) \\ &\wedge(f_{i*n+m-1} \vee f_{(i+1)*n+m-1} \vee \cdots \vee f_{(k-1)*n+m-1})\end{aligned}$$

If both $\mathcal{M}_k^{t_m}$ and ${}_i\mathcal{L}_k^{t_m}$ are satisfied by $S_0, \cdots, S_{n*k}$, then $S_{i*n}S_{i*n+1}\cdots S_{k*n}$ comprises a loop in which $S_j \overset{t_m}{\to} S_{j+1}$ for some j such that $i*n \leq j < k*n$. On the other hand, if a computation $M_0 \cdots M_k$ exists such that M_i is identical to M_k and $M_j \overset{t_m}{\to} M_{j+1} (i \leq j < k)$, then $\mathcal{M}_k^{t_m} \wedge {}_i\mathcal{L}_i^{t_m}$ can be satisfied by assigning as follows: (i) for $0 \leq j \leq k$, $S_{j*n}, S_{j*n+1}, \cdots, S_{j*n+i_j-1} = M_j$ and $S_{j*n+i_j}, \cdots, S_{(j+1)*n} = M_{j+1}$, and (ii) for j $(i \leq j < k)$, if $M_j \overset{t_m}{\to} M_{j+1}$, then $f_{j+n+m-1} = 1$; otherwise $f_{j+n+m-1} = 0$.

As a result, we obtain:

$$\varphi_k^{t_m} := \mathcal{M}_k^{t_m} \wedge \bigvee_{0 \leq i \leq k-1} {}_i\mathcal{L}_k^{t_m}$$

Because of the above discussions, one can see that if $\varphi_k^{t_m}$ is satisfiable, then transition t_m is $L3$-live; otherwise, no computation of length less than or equal to k exists that is a witness for $L3$-liveness of t_m.

5 Transition Ordering

In this section, we introduce an important pre-processing procedure for our method. Thus far we have not discussed the order of the transitions; we have implicitly assumed that transitions with small indices are taken first into consideration in constructing a formula to be checked. However the state space that can be explored by our method critically depends on this order of transitions. As an example, take the Petri net shown in Figure 1 again. As stated before, all reachable state can be explored by our method even with $k = 1$ if transitions are considered in the order $t_1, t_2, \cdots, t_7$. Now suppose that the order were the opposite, that is, $t_7, t_6, \cdots, t_1$. In this case the number of states that can be checked would be considerably reduced. Precisely, when $k = 1$, the state sets that can be reached would be $\{M_0, M_1, M_2\}$.

To obtain an appropriate order, we develop a heuristic as shown in Figure 4. In this algorithm a set *Visited* and a queue *Done* are used to represent the set of already visited places and the order of the transitions, respectively. This algorithm traverses the net structure in a depth first manner, from a place with a token in the initial state M_0. The procedure *visit_place*() is called with the parameter p being that place. In the procedure, p is added to *Visited* first. Then for each of the output transitions, say t, the following is done: If t has not yet been ordered and all its input places have been visited, then t is enqueued and the procedure is recursively called with the parameter being each of t's output places. Because a transition t is ordered after all its input places are visited, t is usually ordered earlier than the transitions in ${}^\bullet({}^\bullet t)$.

Transitions that can not be reached from the initially marked places are not ordered in this algorithm. Since a safe Petri net has no source transition, those transitions will never be enabled and thus can be safely ignored.

Finally we should remark that a problem similar to this transition ordering arises in using *transition chaining* [14], a technique for speeding up BDD-based reachability analysis. The authors of [14] suggested that given a transition t, all transitions in $^{\bullet}(^{\bullet}t)$ should be ordered first. However no detailed algorithm was presented in their paper.

```
main {
set $Visited := \emptyset$;
queue $Done := \emptyset$;
for $p \in P_0$        // $P_0$ is the set of marked places in $M_0$.
call $visit\_place(p)$;
}

$visit\_place(p)${
add $p$ to $Visited$;
for $t \in p^{\bullet}$
if($t \notin Done$ and $\forall \hat{p} \in {}^{\bullet}t\ [\hat{p} \in Visited]$){
enqueue $t$ to $Done$;
for $\hat{p} : \hat{p} \in t^{\bullet}$ and $\hat{p} \notin Visited$
call $visit\_place(\hat{p})$;
}
}
```

Fig. 4. Algorithm for ordering transitions.

6 Experimental Results

We conducted experimental evaluation using a Linux workstation with a 2GHz Xeon processor and 512MByte memory. All Petri nets used in the experiment were taken from [8]. Table 1 shows the size of these Petri nets. Remember that $m = |\mathcal{P}|$ is the number of places and $n = |\mathcal{T}|$ is the number of transitions. The 'States' column represents the number of reachable states reported in [8]. These Petri nets all contain deadlock.

We checked the following properties: (1) L1-liveness for a transition; (2) deadlock freedom; and (3) L3-liveness for a transition. For (1) and (3), the transition to be checked was selected at random. In the experiment, ZChaff, an implementation of Chaff [13], was used as a SAT solver. *Structure preserving transformation* [15] was used for transforming the formula to be checked into CNF, as ZChaff only takes a CNF Boolean formula as an input.

For each model we incremented k until a satisfying assignment (that is a counterexample or a witness) was found. Table 2 shows the time (in seconds)

Table 1. Problem Instances.

Problem	m	n	States
DARTES(1)	331	251	>1500000
DP(12)	72	48	531440
ELEV(1)	63	99	163
ELEV(2)	146	299	1092
ELEV(3)	327	783	7276
ELEV(4)	736	1939	48217
HART(25)	94	92	>1000000
HART(50)	252	152	>1000000
HART(75)	377	227	>1000000
HART(100)	502	302	>1000000
KEY(2)	94	92	536
KEY(3)	129	133	4923
KEY(4)	164	174	44819
MMGT(2)	86	114	816
MMGT(3)	122	172	7702
MMGT(4)	158	232	66308
Q(1)	163	194	123596

required by ZChaff to find a satisfying assignment for the value of k. For some cases, a counterexample/witness could not be found because no such computation existed, processing was not completed within a reasonable amount of time, or memory shortage occurred. For these cases we show the largest k for which our method was able to prove that no satisfying assignment exists (denoted as '$> k$') and the time used to prove it (denoted with parentheses).

For comparison purposes, the following two other methods were tested: ordinary SAT-based bounded model checking and BDD-based model checking. Both methods are implemented in the NuSMV tool [4]. We used the PEP tool [7] to generate the input programs to NuSMV from the Petri nets. Table 2 shows the results of using these methods; the "SAT" columns and the "BDD" columns show the results of using ordinary SAT-based bounded model checking and those of BDD-based model checking, respectively. $L3$-liveness was not checked because this property is not always verifiable with these methods.

From the results, it can be seen that the proposed method completed verification for most of the problems that the two existing methods were not able to solve. When compared to the ordinary bounded model checking, one can see that our method required a much smaller value of k to find a witness. As a result, even for the problems the existing methods were able to handle (e.g., DP(12) or KEY(2)), our method often outperformed these methods in execution time.

Table 2. Results.

	$L1$-Liveness					Deadlock					$L3$-Liveness	
	Ours		NuSMV			Ours		NuSMV			Ours	
			SAT		BDD			SAT		BDD		
Problem	k	Time	k	Time	Time	k	Time	k	Time	Time	k	Time
DARTES(1)	1	0.01	>1	0.01	NA	1	0.02	>1	0.02	NA	>10	0.18
DP(12)	1	0.0	3	0.01	NA	1	0.0	12	37308.6	NA	1	0.0
ELEV(1)	2	0.01	8	1.14	0.46	1	0.04	9	0.26	0.55	>27	0.19
ELEV(2)	2	0.03	>5	1.75	8.72	1	1.12	>5	6.82	32.46	>8	0.14
ELEV(3)	2	0.11	>1	0.0	NA	1	14.64	>1	0.02	NA	>3	0.05
ELEV(4)	>1	0.07	>0	0.0	NA	1	208.4	>0	0.01	NA	>1	0.0
HART(25)	1	0.0	>6	0.47	NA	1	0.0	>6	5.67	NA	>38	1022.46
HART(50)	1	0.0	>2	0.02	NA	1	0.0	>2	0.02	NA	>19	4.47
HART(75)	1	0.0	>1	0.0	NA	1	0.0	>1	0.01	NA	>12	158.68
HART(100)	1	0.01	>0	0.0	NA	1	0.0	>0	0.0	NA	>9	34.11
KEY(2)	1	0.0	6	0.03	746.07	1	0.0	>9	1.61	NA	>29	0.29
KEY(3)	1	0.01	6	0.04	NA	1	0.0	>6	1.43	NA	>20	0.4
KEY(4)	1	0.01	>4	0.14	NA	1	0.0	>4	0.18	NA	>15	0.28
MMGT(2)	1	0.0	5	0.03	0.65	1	0.0	8	0.07	0.71	>22	0.26
MMGT(3)	2	0.01	>6	2.81	1.49	2	0.07	>6	46.78	1.6	>14	0.18
MMGT(4)	3	0.02	>5	0.24	3.53	3	0.25	>5	12.64	3.61	>10	0.18
Q(1)	>13	5.28	>4	0.19	NA	1	0.02	>4	0.49	NA	>13	3.3

7 Conclusions

In this paper, we proposed a new method for bounded model checking. By exploiting the interleaving nature of Petri nets, our method generates much more succinct formulas than ordinary bounded model checking, thus resulting in high efficiency. We applied the proposed method to a collection of safe Petri nets. The results showed that our method outperformed ordinary bounded model checking in all cases tested and beat BDD-based model checking especially when a short computation exists that was a counterexample/witness.

There are several directions in which further work is needed. First a more comprehensive comparison is needed with existing verification methods other than those discussed here. Especially, comparison with bounded reachability checking proposed by Heljanko [8] should be conducted, since his approach is similar to ours in the sense that both can be used for reachability checking for safe Petri nets. Other important verification methods include those based on partial order reduction.

We think that applying the proposed method to other models than (pure) Petri nets is also important. To date we have obtained some results of applying our method to detection of feature interactions in telecommunication services [17]. Hardware verification (e.g., [10,18]) is also an important area where the applicability of our method should be tested.

Another direction is to extend the method to verify properties other than reachability and L3-liveness. Currently we are working on modifying the proposed method for verification of arbitrary LTL$_{-X}$ formulas. LTL$_{-X}$ is an important class of temporal logic; many model checking tools, such as SPIN [11] and the one described in [9], can be used to verify temporal formulas in this class.

Recently it was shown that SAT can be used, in combination with unfolding [12], for coverability checking of unbounded Petri nets [1]. Our approach can not be directly applied to unbounded Petri nets; but we think that extending the proposed method to infinite state systems is a challenging but important issue.

Acknowledgments. The authors wish to thank anonymous referees for their useful comments.

References

1. P. A. Abdulla, S. P. Iyer, and A. Nylén. Sat-solving the coverability problem for Petri nets. *Formal Methods in System Design*, 24(1):25–43, 2004.
2. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), LNCS 1579*, pages 193–207. Springer-Verlag, 1999.
3. A. Cheng, J. Esparza, and J. Palsberg. Complexity results for 1-safe nets. *Theoretical Computer Science*, 147(1-2):117–136, 1995.
4. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.
5. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.
6. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of 13th International Computer Aided Verification Conference (CAV'01), LNCS 2102*, pages 436–453. Springer-Verlag, 2001.
7. B. Grahlmann. The PEP Tool. In *Proceedings of 9th Computer Aided Verification (CAV'97), LNCS 1254*, pages 440–443. Springer-Verlag, June 1997.
8. K. Heljanko. Bounded reachability checking with process semantics. In *Proceedings of the 12th International Conference on Concurrency Theory (Concur'2001), LNCS 2154*, pages 218–232, Aalborg, Denmark, August 2001. Springer.
9. K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4&5):519–550, 2003.
10. P.-H. Ho, A. J. Isles, and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 529–536. ACM Press, 1998.
11. G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
12. K. L. McMillan. A technique of state space search based on unfolding. *Form. Methods Syst. Des.*, 6(1):45–65, 1995.

13. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient sat solver. In *Proceedings of 39th Design Automation Conference*, pages 530–535. ACM Press, 2001.
14. E. Pastor, J. Cortadella, and O. Roig. Symbolic analysis of bounded petri. *IEEE Transactions on Computers*, 50(5):432–448, May 2001.
15. D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, September 1986.
16. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *Proc. of International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000), LNCS 1954*, pages 108–125. Springer-Verlag, 2000.
17. T. Tsuchiya, M. Nakamura, and T. Kikuno. Detecting Feature Interactions in Telecommunication Services with a SAT solver. In *Proc. of 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)*, pages 131–134. IEEE CS Press, 2002.
18. T. Yoneda, T. Kitai, and C. Myers. Automatic derivation of timing constraints by failure analysis. In *Proceedings of the 14th International Conference on Computer-aided Verification (CAV 2002), LNCS vol.2404*, pages 195–208. Springer-Verlag, 2002.