On Model Checking for Petri Nets and a Linear-Time Temporal Logic [*]

Tomohiro Yoneda
Tokyo Institute of Technology
yoneda@cs.titech.ac.jp

Holger Schlingloff
Technische Universität München
schlingl@informatik.tu-muenchen.de

Abstract

This paper presents an efficient model checking algorithm for Petri nets. It is based on the reduced state space generation where the result of the evaluation on the full state space and the reduced state space is identical. This reduction of the state spaces is possible, because (1) the firings of the transitions are only partially ordered by causality and a given formula, and (2) the order of firings of transitions not related by this partial order is irrelevant for the evaluation of the given formula. For the considerable reduction of the state spaces, a linear-time temporal logic without the next-time operator is used. The concrete algorithm of the reduced state space generation is shown as well as the brief sketch of the correctness proof. Finally, some experimental results are shown to demonstrate the efficiency of the proposed method.

1 Introduction

Petri nets have been widely studied as a formalism for concurrent and distributed systems. Especially they proved to be appropriate for describing the behavior of asynchronous circuits[DNS90], which have become recently of increasing interest because of certain limitations in synchronous circuits. Once such systems are modeled by Petri nets, they are open to formal verification using automated tools. Reachability analysis can be used to check

[*]Presented at IEICE FTS92 workshop, Kyoto, Japan, June 1992

certain classes of safety properties. Model checking techniques [CES86] allow to check more general classes of properties (represented by temporal logic formulas) including both safety and liveness requirements.

Unfortunately, even though the number of reachable states is in many cases finite, in most practical systems it is often extremely large. This so called *state explosion* problem has been one of the biggest bottlenecks for the automatic verification of practical systems. Recently, several research projects to avoid this problem are reported. One approach is *symbolic model checking*, in which sets of states are represented symbolically and the set that satisfies a formula is computed as the fixpoint of a certain functional. Implementations of this method with binary decision diagrams (BDDs) succeed in handling huge number of states [BCD$^+$90]. However, when the state space is rather sparse (as it is often the case with models of asynchronous circuits), BDD based techniques do not seem to represent the state space efficiently without pre-analysis or variable re-ordering by hand. In this sense, this approach is not fully automatic. Another approach to confine the state explosion problem is to use partial orders. These techniques are based on the fact that not all possible interleavings of concurrent events are relevant for the properties to be checked. Several methods [God90, McM92] based on partial orders have been proposed for the efficient reachability analysis of Petri nets. In [GW91], the *trace automaton method* of [God90] is extended for checking properties of extended linear temporal logic on systems of communicating Büchi automata. Another method, using *stubborn sets*, was developed in [Val90] for the efficient model checking of linear-time temporal logics on variable/transition systems. Independently, in [YTK91], a similar idea for the zero-reachability problem of time Petri nets has been presented.

In this work, mainly for the verification of asynchronous circuits, we aim at the development of an efficient model checking algorithm for one-safe Petri nets and linear-time temporal logic (LTL). The relation between our and the above mentioned methods is as follows:

- Our loop condition differs from the ones found in [Val90] and [GW91], resulting in an in many cases smaller state space.

- We focus on only *just* runs rather than unrestricted runs or general liveness conditions, because it is suitable to model asynchronous circuits. This restriction both simplifies the proof of correctness for our method and makes the algorithm efficient.

- Our algorithm works on-line, *i.e.* runs are checked for satisfiability while they are constructed.
- We give a concrete algorithm for a specific model of computation and a fixed logic. We also show how to apply our method to the verification of asynchronous circuits and give some experimental results.

The rest of this paper is organized as follows. In the next section, definitions of Petri nets and their runs are given. In section 3, we briefly introduce the logic we are using. The partial order model checking algorithm is presented in section 4. Some experimental results are shown in the next section, and finally we summarize our discussion.

2 Petri nets

A *Petri net* N is a tuple $N = (P, T, F, s_0)$, where

1. P is a finite set of *places*,
2. T is a finite set of *transitions* $(P \cap T = \emptyset)$,
3. $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*, and
4. $s_0 \subseteq P$ is the *initial state* of the net.

For any transition t, $\bullet t = \{p \mid (p,t) \in F\}$, $t\bullet = \{p \mid (t,p) \in F\}$ denote the *preset* and the *postset* of t, respectively. For simplicity, we disallow "idle" transitions t with $\bullet t = t\bullet$.

A *state* s of N is any subset of P. If $p \in s$, we say that p is *occupied*, else p is *empty* at s.

A transition t is *enabled* at state s if $\bullet t \subseteq s$ (all its input places are occupied at s). Otherwise, the transition is *disabled.*

A state s' is the *result of firing* a transition t at a state s ($s \xrightarrow{t} s'$), if t is enabled at s and $s' = (s - \bullet t) \cup t\bullet$. s' is a *successor state* of s ($s \rightarrow s'$), if there exists some t such that $s \xrightarrow{t} s'$. A *run* $\rho = (s_0 s_1 s_2 \cdots)$ of N is a finite or infinite sequence of states s_i such that s_0 is the initial state of N, and for every i $s_i \rightarrow s_{i+1}$. Note that with this definition we adopt the so-called *interleaving view*: The parallel execution of two independent transitions t_1 and t_2 (i.e., $(\bullet t_1 \cup t_1\bullet) \cap (\bullet t_2 \cup t_2\bullet) = \emptyset$) at a state s_i is modelled by the two sequences $(...s_i \xrightarrow{t_1} s_{i+1} \xrightarrow{t_2} s_{i+2}...)$ and $(...s_i \xrightarrow{t_2} s'_{i+1} \xrightarrow{t_1} s_{i+2}...)$.

A state s is *reachable* if there exists a finite run whose last state is s. A Petri net is *one-safe* if for all reachable states s and all $t \in T$, $\bullet t \subseteq s$ implies $t\bullet \cap s \subseteq \bullet t$ (i.e., if t is enabled, then all output places of t which are not input places of t are empty). In the sequel a *net* will always be a one-safe Petri net. It is very easy to extend our theory to elementary nets, *i.e.* nets in which a transition t is enabled at state s iff $\bullet t \subseteq s$ and $t \bullet \cap s \subseteq \bullet t$.

A transition t is said to be *continuously enabled* in a run ρ, if there exists an i such that t is enabled at all states s_j of ρ with $j \geq i$. A run ρ is *just* if there is no continuously enabled transition in ρ, *i.e.* if for any t and any i there exists a state s_j $(j \geq i)$ such that t is disabled in s_j. Note that every just run ρ is *maximal*, that is, if ρ is finite, then its final state contains no enabled transitions.

The *behavior* $B(N)$ of N is the set of all just runs of N.

Here, we add some intuitive explanation about the maximality and the justice of runs. If we did not to require the maximality of runs, then every invariance properties of the kind "there exists a run such that every state satisfies p" would be trivially true or false, depending on whether the initial state satisfies p or not. Justice means that not only the whole net, but every parallel component of the net is executed whenever possible. Let's consider the net shown in Figure 1 representing that *action 1/action 2* and *action 3/action 4* are repeatedly activated in independent processes A and B, respectively. Suppose that the initial state is $s_0 = \{a, c\}$. A run $\rho = (s_0 s_1)^\omega$ (*i.e.* $\rho = (s_0 s_1 s_0 s_1 \cdots)$) of the net is not just, where $s_1 = \{b, c\}$, because t_3 is continuously enabled in ρ. Thus, the proposition "*action 4* is eventually activated" always holds also in this net. Since the behavior of each process is independent, this restriction of runs is also reasonable.

Finally, we'll show how nets are suitable to model asynchronous or speed independent circuits. Asynchronous circuits are those which function correctly assuming arbitrary delays in the components and no delays in the wires[DNS90]. Those circuits can be modeled with nets as follows. Each wire of the circuits takes two values, 0 and 1. Thus, it is represented by two places, either of which is always occupied. A token in those places are controlled by a component according to the function of the component and the state of its input wires. Figure 2 illustrates a net modelling a two input AND gate as an example. In this figure, p_1 and p_2 represent an input line of the AND gate, p_3 and p_4 represent another input line, and p_5 and p_6 represent an output line. The change of the output value from 0 to 1 is represented by firing the transition t_1, while the change from 1 to 0 is represented by firing t_2 or t_3. Although each place has several input/output

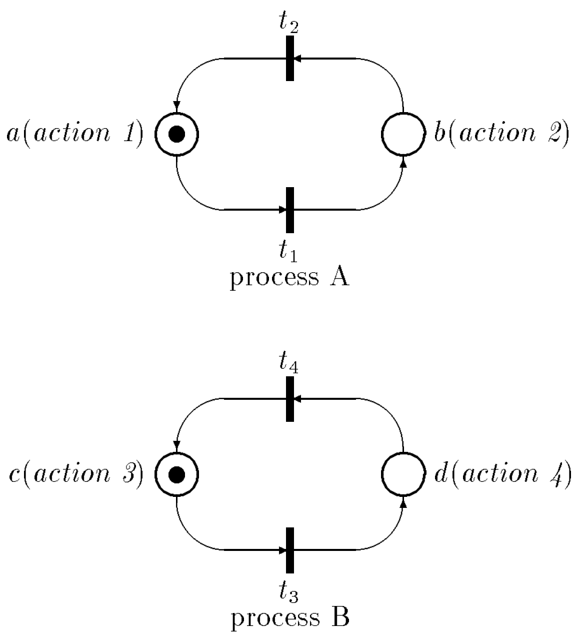

Figure 1: A net representing two independent processes.

transitions, only p_5 and p_6 are controlled by t_1, t_2, and t_3. The firing of, for example, t_1 does not move any token in p_1 or p_3. The justice of runs completely represent arbitrary delays in the components.

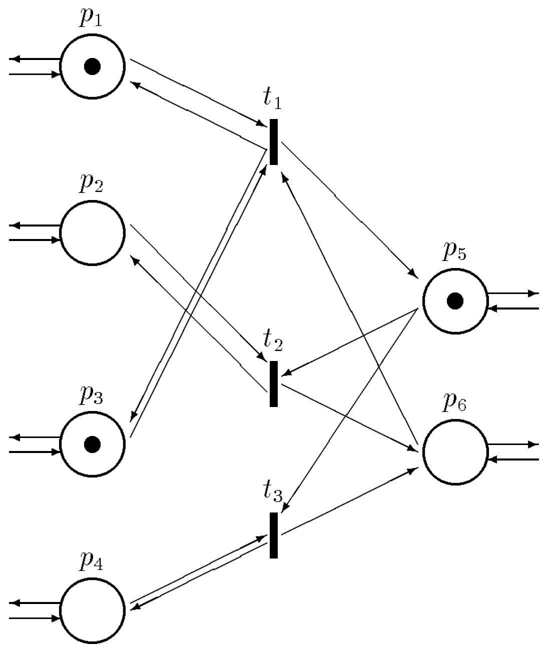

Figure 2: A net modelling a two input AND gate.

3 Logic

To obtain an efficient model checking algorithm, we want to reduce the size of the state space, using the fact that firings of independent transitions are only *partially ordered* by causality and a given property, and that the order of firings of transitions not related by this partial order is irrelevant for the evaluation of the given property. Of course, the result of the evaluation on both the full state space and the reduced state space must be identical. If logics have some properties, then this property may hold on considerably reduced state space. For example, in branching-time temporal logics such as CTL[CES86], branches in the state space have significant meanings than in linear-time temporal logics. Thus, in order to preserve the truth of the formulas, the reduced state space must keep every branch of the full state space, but this limits the reduction of the state space. Further, if a formula includes the next-time operator, then it is rather clear that only the full state space gives the correct truth of the formula. Therefore, we consider a linear-time temporal logic without the next-time operator, which is defined formally below.

Let for the rest of this section a net $N = (P, T, F, s_0)$ be given. The set of *atomic propositions* for N is the set P of places. The formulas of linear temporal logic (LTL) for N are defined inductively as follows.

1. Every atomic proposition $p \in P$ is a formula,
2. *false* is a formula, and
3. If φ_1 and φ_2 are formulas, then $(\varphi_1 \rightarrow \varphi_2)$ and $(\varphi_1 \mathcal{U} \varphi_2)$ are formulas

Additional boolean connectives *true*, $\wedge$, $\vee$, $\leftrightarrow$, and temporal connectives $\Box$, $\Diamond$ can be defined as usual.

Validity (*i.e.* $(\rho, i) \models \varphi$) of an LTL formula φ in a run $\rho = (s_0 s_1 s_2 \cdots)$ and a state $s_i \in \rho$ is defined by induction on φ:

1. $(\rho, i) \not\models$ *false*
2. $(\rho, i) \models p$ iff $p \in s_i$ for $p \in P$
3. $(\rho, i) \models (\varphi_1 \rightarrow \varphi_2)$ iff $(\rho, i) \models \varphi_1$ implies $(\rho, i) \models \varphi_2$
4. $(\rho, i) \models (\varphi_1 \mathcal{U} \varphi_2)$ iff there exists $j \geq i$ such that $s_j \in \rho$, $(\rho, j) \models \varphi_2$, and for all k, $i \leq k < j$, $(\rho, k) \models \varphi_1$

φ is *valid* in ρ ($\rho \models \varphi$) if $(\rho, 0) \models \varphi$. φ is *valid* in N ($N \models \varphi$) if φ is valid in all $\rho \in B(N)$. φ is *satisfiable* in N if $N \not\models \neg\varphi$, i.e. iff there exists a run $\rho \in B(N)$ such that $\rho \models \varphi$.

Note that our definition of the temporal operator $\mathcal{U}$ is *stuttering invariant*, that is, the next-time operator is not definable in our logic, and that we use the so called *initial* definitions of satisfiability and validity.

4 Partial order model checking

In this section, we first present an LTL model checking algorithm, and then show how to generate a reduced state space for the given net and formula. Let for the rest of this section φ be a fixed LTL formula for the net $N = (P, T, F, s_0)$.

4.1 LTL model checking algorithm

Suppose that a relation $R \subseteq 2^P \times 2^P$ represents a state space, where 2^P denotes the set of all subsets of P. For example, R for a full state space of N is defined as : $(s, s') \in R$, iff s is reachable from s_0 and $s \rightarrow s'$.

The following model checking algorithm of LTL is based on [LP85].

Let U_φ be the set of formulas of the form $(\psi_1 \mathcal{U} \psi_2)$ appearing in φ. A *guess* g is any subset of U_φ. The intuition behind this definition is that a guess contains exactly those $\mathcal{U}$-formulas which are guessed to be true in a given state.

Given a state s and a guess g, the truth value of the subformula ψ of φ in (s, g) (*i.e.* $(s, g)||\!-\psi$) can be determined inductively as follows:

1. if $\psi = \mathit{false}$, then $(s, g) \not||\!-\psi$
2. if $\psi \in P$, then $(s, g)||\!-\psi$ iff $\psi \in s$
3. if $\psi = (\psi_1 \rightarrow \psi_2)$, then $(s, g)||\!-\psi$ iff $(s, g)||\!-\psi_1$ implies $(s, g)||\!-\psi_2$
4. if $\psi = (\psi_1 \mathcal{U} \psi_2)$, then $(s, g)||\!-\psi$ iff $\psi \in g$

A guess g is called *consistent* with state s, if for every formula $\psi = (\psi_1 \mathcal{U} \psi_2) \in U_\varphi$, the following holds:

1. if $\psi \in g$, then $(s, g)||\!-\psi_1$ or $(s, g)||\!-\psi_2$
2. if $(s, g)||\!-\psi_2$, then $\psi \in g$

An *atom* is a pair $\alpha = (s, g)$ such that g is a guess consistent with state s. The set of *initial atoms* consists of all atoms $\alpha = (s_0, g)$ such that s_0 is the initial state of the given net N, and $\alpha||\!-\varphi$, where φ is the given formula. Note that there are at most $2^{|P|} \cdot 2^{|U_\varphi|}$ different atoms for any net N and formula φ.

Let α be an atom. The set of *positive future obligation* $PFO(\alpha) \subseteq U_\varphi$ is given by

$$PFO(\alpha) = \{\psi \mid \psi = (\psi_1 \mathcal{U} \psi_2),\ \alpha||\!-\psi, \alpha \not||\!-\psi_2\}.$$

The set of *negative future obligation* $NFO(\alpha) \subseteq U_\varphi$ is given by

$$NFO(\alpha) = \{\psi \mid \psi = (\psi_1 \mathcal{U} \psi_2), \alpha \not||\!-\psi,\ \alpha||\!-\psi_1\}.$$

Thus, the positive future obligations of an atom α are all those $\mathcal{U}$-formulas which are guessed to be true but are not immediately satisfied by α, and the negative future obligations are those $\mathcal{U}$-formulas which are guessed to be false but cannot be refuted by looking at α alone.

Now, we'll define an atom graph, on which the model checking algorithm traverse, by a relation S such that $(\alpha_1, \alpha_2) \in S$, iff

1. $\alpha_1 = (s_1, g_1)$ and $\alpha_2 = (s_2, g_2)$ are atoms, and
2. $(s_1, s_2) \in R$, and
3. $PFO(\alpha_1) \subseteq g_2$, and
4. $NFO(\alpha_1) \cap g_2 = \emptyset$.

Let S^* be the transitive reflexive closure of S. A nonempty set of atoms $\mathcal{G}$ is said to form a *strongly connected component* (SCC) if $(\mathcal{G} \times \mathcal{G}) \subset S^*$. By abuse of notation we say that $(\alpha, \mathcal{G}) \in S^*$ if there exists an $\alpha' \in \mathcal{G}$ such that $(\alpha, \alpha') \in S^*$.

An SCC $\mathcal{G}$ is *maximal* if there do not exist two different atoms α_1, α_2 such that $(\alpha_1, \alpha_2) \in S^*$, $(\alpha_2, \alpha_1) \in S^*$, $\alpha_1 \in \mathcal{G}$ and $\alpha_2 \notin \mathcal{G}$. Note that for every atom α there exists exactly one maximal SCC $\mathcal{G}$ such that $\alpha \in \mathcal{G}$ ($\mathcal{G}$ may be $\{\alpha\}$).

An SCC $\mathcal{G}$ is said to be *self-fulfilling* if for every atom $\alpha_1 \in \mathcal{G}$ and for every formula $\psi = (\psi_1 \mathcal{U} \psi_2) \in U_\varphi$ such that $\alpha_1 || \!- \psi$, there exists an atom $\alpha_2 \in \mathcal{G}$ such that $\alpha_2 || \!- \psi_2$.

A SCC $\mathcal{G}$ is called *just* if there is no continuously enabled transition in $\mathcal{G}$.

A SCC $\mathcal{G}$ is called *accepting* if it is self-fulfilling and just.

Theorem 4.1 : Let N be the given net and φ be the given LTL formula for N. φ is satisfiable by N, iff there exists an initial atom α and an accepting SCC $\mathcal{G}$ such that $(\alpha, \mathcal{G}) \in S^*$. □

Theorem 4.2 : For any accepting SCC, there exists an accepting maximal SCC. □

Therefore, in order to find out whether φ is satisfiable by N it is sufficient to generate the maximal SCC's reachable from any initial atom and to check whether they are accepting. Note that this check can be done on-line, that is, we can enumerate the maximal SCC's while constructing the atom graph. But, for simplicity, we don't mention more about it here.

The complexity of the LTL model checking is exponential in the size of a formula[LP85], linear in the size of the state graph[LP85], and exponential in the size of a net.

4.2 Reduced state space generation

Here, we try to generate the reduced state space of a given net with respect to a given formula on which the LTL model checker still yields the correct result.

Usually, a given formula contains only a small subset of P. Let P_φ denote the smallest subsets of P such that all atomic propositions appearing in φ are contained in P_φ.

We consider a set of enabled transitions whose firing orders are relevant when an enabled transition t_f fires. We call this set *the dependent set of* t_f. Basically, conflicting transitions with t_f should belong to the dependent set of t_f. However, a problem occurs when some of the conflicting transitions are disabled. In the example of Figure 3, although t_2 is disabled, it may conflict with t_1 when it gets enabled. Thus, we include in the dependent set the enabled transitions (in this case, t_4) that can enable such t_2. There is another class of relevant transitions. If t_f affects some of P_φ, the firing order with other transitions that also affect some of P_φ is relevant to evaluate the given formula. For example, t_1 affects both p_1 and p_2 and t_7 affects p_8 in the net of Figure 3. If the given formula is $\diamond(p_2 \wedge p_7 \wedge \neg p_8)$, then the firing order between t_1 and t_7 is relevant. We call these transitions that affects some of P_φ *visible*. If t_f is visible, then other enabled visible transitions should be included in the dependent set of t_f. Further, we have to treat disabled visible transitions in the same manner as disabled conflicting transitions. In the same example as above, since a visible transition t_6 is disabled, t_8 that can enable t_6 should be included in the dependent set. This consideration means that we can identify visible transitions with conflicting transitions. Finally, the dependent set of t_1 in this case is $\{t_1, t_4, t_7, t_8\}$.

Now, when we fire t_f, the other transitions in the dependent set of t_f should be fired. That is, if t' is included in the dependent set of t_f, then the dependent set of t' should also be fired. In the above example, the dependent set of t_4 includes t_5, and the dependent set of t_8 includes t_9. Since no new transitions are introduced by t_5 and t_9, we can obtain a set F $= \{t_1, t_4, t_5, t_7, t_8, t_9\}$ starting from t_1. F has the property that the dependent set of any element of F is also in F. We call this F the *firable set* of the considered state. The reduced state space is constructed by firing each transitions in the firable set instead of every enabled transition. Note that the firable set of a state depends on the first chosen transition. For example, if we first choose t_4, then the firable set is $\{t_4, t_5\}$. Although any firable set works correctly, the smallest firable set may generate a small state space.

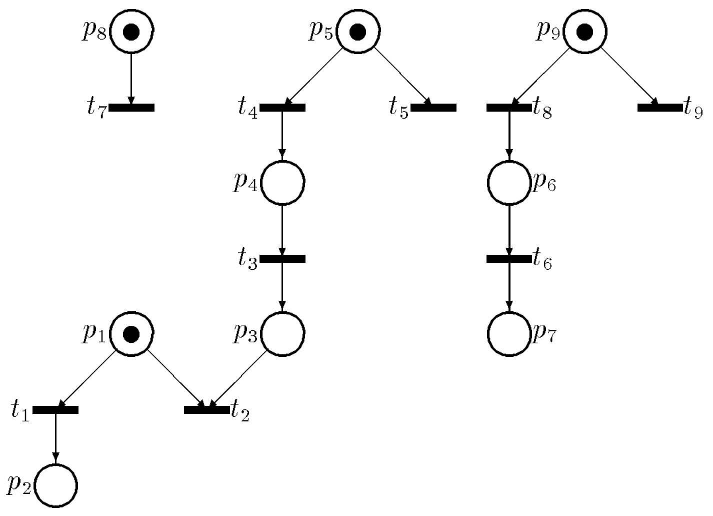

Figure 3: An example including disabled visible or disabled conflicting transitions.

This idea is formally stated below.

Definition 4.1 : For any transition t, state s, and set of transitions M, we define the following sets of transitions:

- *choose*(M) chooses one element of M by a fixed rule. For example, we can choose a smallest-indexed transition. Note that the correctness of our method does not depend on this function. It only influences the efficiency.

- *necessary*$(t, s) = \{t' \mid p = \mathit{choose}((P - s) \cap \bullet t), p \in t'\bullet\}$.
  necessary(t, s) is a set of transitions which are necessary for the firing of t: If t is enabled in a state s, then *necessary*(t, s) is empty, and if t is disabled in s then there exists an empty place $p \in \bullet t$ such that all t' with $p \in t'\bullet$ are contained in *necessary*(t, s). For example, *necessary*$(t_2, s) = t_3$ in Figure 3, where s is the state shown in the figure.

- *necessary*$^*(t, s)$ is the smallest set M containing t such that for any

$t' \in M$, *necessary*$(t', s) \subseteq M$. If t is enabled, then $N^*(t, s) = \{t\}$. For example, *necessary*$^*(t_2, s) = \{t_3, t_4\}$ in Figure 3.

- *enabled*$(s) = \{t \mid \bullet t \subseteq s\}$
  enabled(s) is the set of enabled transitions at s.

- *active*$(t, s) = \{t' \mid t' \in$ *necessary*$^*(t, s) \cap$ *enabled*$(s)\}$
  active(t, s) is the set of enabled transitions in *necessary*$^*(t, s)$. Note that *active*$(t, s) = \{t\}$, if t is enabled at s.

- *visible*$_\varphi = \{t \mid (\bullet t \cup t\bullet) \cap P_\varphi \neq \emptyset\}$
  visible$_\varphi$ contains all transitions which affect places appearing in φ.

- *conflict*$(t) = \{t' \mid \bullet t \cap \bullet t' \neq \emptyset\}$
  conflict(t) is the set of all transitions statically conflicting with t. Note that t is also in *conflict*(t).

- $\mathit{conflict}^+(t) = \begin{cases} \mathit{conflict}(t) & \text{if } t \notin \mathit{visible}_\varphi \\ \mathit{conflict}(t) \cup \mathit{visible}_\varphi & \text{otherwise} \end{cases}$
  conflict$^+(t)$ is the set of transitions whose firing orders with t are basically relevant for the evaluation of φ.

- $\mathit{dependent}(t, s) = \bigcup_{t' \in \mathit{conflict}^+(t)} \mathit{active}(t', s)$

□

Now, let *firable*(s) be any set satisfying:

1. for any $t \in$ *firable*(s), *dependent*$(t, s) \subseteq$ *firable*(s), and
2. for any enabled transition t at s, there exists a state s' such that $s = s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n = s'$, for $1 \leq i \leq n$, $t_i \in$ *firable*(s_i), and $t \in$ *firable*(s').

Then, the reduced state space is defined by the smallest relation R satisfying :

1. $(s_0, s') \in R$, iff for some t, $s_0 \xrightarrow{t} s'$ and $t \in$ *firable*(s_0), and
2. $(s, s') \in R$, iff there exist s'' and t such that $(s'', s) \in R$, $s \xrightarrow{t} s'$, and $t \in$ *firable*(s).

The second requirement for the firable sets is necessary to give the chances for firing continuously enabled transitions. For example, let's consider the net shown in Figure 4 with a visible transition t_1 and invisible transitions t_2 and t_3. Let $s_0 = \{a, b\}$ (the initial state), $s_1 = \{a, d\}$, $s_2 = \{b, c\}$, and $s_3 = \{c, d\}$. Without the second requirement, its reduced state space may be

$$(s_0, s_1) \in R$$
$$(s_1, s_0) \in R,$$

because it is possible that the firable sets are $\mathit{firable}(s_0) = \{t_2\}$ and $\mathit{firable}(s_1) = \{t_3\}$. Since t_1 is continuously enabled, this reduced state space does not yield the same result as the full state space in the evaluation of, for example, $\diamond c$. From the second requirement, the firable set of, for example, s_0 must be $\{t_1, t_2\}$, and this generates the correct reduced state space.

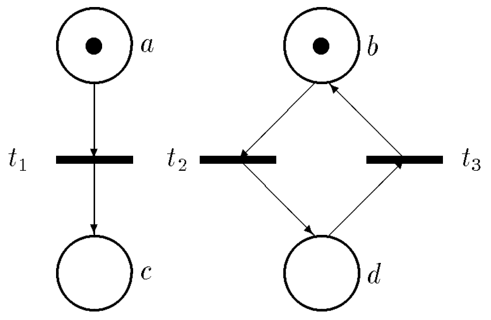

Figure 4: An example of a net.

Figure 5 shows one implementation of our partial order model checking algorithm in the on-line style. Note that $\mathit{dependent}^*(t, s)$ is the smallest set M containing t such that for any $t' \in M$, $\mathit{dependent}(t, s) \in M$. The second requirement for the firable sets is implemented by extending the firable set when continuously enabled transitions are found at the root of a SCC. It uses the algorithm to find SCC's in [AHU74].

Finally, we mention how to handle unrestricted runs. Again, consider the net shown in Figure 4 with a visible transition t_1 and invisible transitions

```
procedure PartialOrderModelChecking(atom a)
var dfnumber, sum, new, succ;
  if (table[a] = undefined) {
    dfnumber := state_number++;
    table[a] := dfnumber;
    push(stack, a);
    t := "some transition enabled at a.s";
    sum := {};
    repeat
      new := $dependent^*$(t,a.s) $-$ sum;
      sum := sum + new;
      succ := {a' | a'=$\langle$s',g'$\rangle$ is a consistent atom,
          a.s$\xrightarrow{t}$a'.s' for some t$\in$ new}
      for all (a' $\in$ succ) do {
        PartialOrderModelChecking(a');
        table[s] := min(table[a], table[a']);
      }
      if (table[a] = dfnumber) /* a is the root of an SCC */
        SCC := {};
        required := fulfilled := {};
        disabled := {};
        repeat
          b := pop(stack);
          push(SCC, b);
          required := required $\cup\{(p\mathcal{U}q) |$ b $\|\!-(p\mathcal{U}q)\}$;
          fulfilled := fulfilled $\cup\{(p\mathcal{U}q) |$ b $\|\!-q\}$;
          disabled := disabled $\cup$ {t | t is disabled in b.s};
        until (b = a);
        if (required = fulfilled) and (disabled = {t | t$\in$T})
          return with success;   /* f is satisfiable by N */
        cont_enabled := {t | t$\in$T}$-$ disabled;
        while SCC $\neq \emptyset$ {
          b := pop (SCC);
          if (cont_enabled $\subseteq$ sum) {
          then table[b] := MAXINT;
          else push (stack, b);
        }
      t := "some transition in (cont_enabled $-$ sum)";
      }
    until (cont_enabled $\subseteq$ sum);
  }
end of procedure
```

Figure 5: Partial order model checking algorithm.

t_2 and t_3. Unlike the previous example,

$$\begin{aligned}(s_0, s_2) &\in R\\ (s_2, s_3) &\in R\\ (s_3, s_2) &\in R\end{aligned}$$

is also a possible reduced state space. This can be obtained by choosing t_1 instead of t_2 at s_0 to compute *firable*(s_0). Note that from this reduced state space, a non-just run $(s_0 s_1)^\omega$ can not be extracted. Hence, for example, $\neg\diamond c$, which is satisfiable by unrestricted runs, is not considered to be satisfied by this reduced state space. Therefore, for example, the following additional requirement for the stubborn sets is necessary for [Val90].

> If there exists an enabled invisible transition at a state s, then *firable*(s) should contain an enabled invisible transition.

By this requirement, even if we choose t_1 at s_0 in the above example, *firable*(s_0) contains an enabled invisible transition t_2. This results in the reduced state space with a non-just run $(s_0 s_1)^\omega$.

It is clear that this additional requirement makes the firable set bigger, and so generates bigger reduced state space. However, note that we don't have to consider non-just runs to model asynchronous circuits, because arbitrary delays in the components of asynchronous circuits can be modeled by the property of just runs where every enabled transition will eventually fire unless it is disabled.

4.3 Correctness of the partial order model checking

Once a reduced state space is generated as shown in the previous subsection, we can obtain the correct result by running the LTL model checker shown in 4.1 on this reduced state space. In this subsection, we show only the sketch of the correctness proof of this partial order model checking.

In order to show that our method is correct, that is, the truth of a given formula on both the full state space and the reduced state space is identical, it is sufficient to show :

P1: For any just run α of a net N, there exists a just run ρ represented by the reduced state space such that the same set of visible transitions fire in α and ρ, and that the firing order of those visible transition in both α and ρ is identical.

P2: The reverse direction of **P1** holds.

It is rather apparent that the set of runs represented by the reduced state space is a subset of the behavior $B(N)$. Hence, **P2** holds. In the rest of this subsection, we try to prove **P1**.

First, the formal definition of runs represented by the reduced state space will be shown. Let $G(N, \varphi)$ denote the reduced state space of N with respect to φ.

Definition 4.2 : In the first visit to a state s, if t is chosen from enabled transitions at s by the algorithm shown in Figure 5, then

$$\mathit{firable}^0(\mu) = \mathit{dependent}^*(t, \mu).$$

If the i-th extension of the firable set at s occurs and t is chosen from the continuously enabled transitions, then

$$\mathit{firable}^i(\mu) = \mathit{dependent}^*(t, \mu) - \bigcup_{j=0}^{i-1} \mathit{firable}^j(\mu).$$

If the i-th extension of the firable set at s does not occur, then

$$\mathit{firable}^i(\mu) = \emptyset.$$

□

Note that

$$\mathit{firable}(s) = \bigcup_{j=0}^{m} \mathit{firable}^i(s),$$

where m is the number of the extension of the firable set at s.

Definition 4.3 : $\rho = (s_0 s_1 s_2 \cdots)$ is a run *extracted from* $G(N, \varphi)$, if

1. s_0 is the initial state of N, and
2. for all $i \geq 0$, $(s_i, s_{i+1}) \in G(N, \varphi)$, and
3. if ρ is finite and s_n is the last state, then there is no enabled transition at s_n, and
4. for any state s appearing in ρ, the state transition $(s \xrightarrow{t'_{i \bmod (m+1)}} s')$ occurs from the i-th appearance of s in ρ, where for $0 \leq j \leq m$, $t'_j \in \mathit{firable}^j(s)$, and m is the number of the extensions of the firable set at s.

□

The above 4 means that if s appears infinitely often in a run ρ extracted from $G(N, \varphi)$, then ρ should be intuitively

$$(s_0 \cdots s \xrightarrow{t'_0} s'_0 \cdots s \xrightarrow{t'_1} s'_1 \cdots s \xrightarrow{t'_m} s'_m \cdots s \xrightarrow{t'_0} s'_0 \cdots).$$

It is necessary that at least one transition in each $\mathit{firable}^j(\mu)$ $(0 \leq j \leq m)$ fires in order to make ρ just. Since for different j and k, transitions in $\mathit{firable}^j(\mu)$ and $\mathit{firable}^k(\mu)$ are concurrent, we only consider one firing order among those transitions.

Then, we introduce a new notion called *i-transitions* because :

1. It is necessary to distinguish two or more different firings of the same transition in a run, and
2. It simplifies the proof to have the information about the number of firings of the transitions that belong to $\mathit{conflict}^+(t)$.

Definition 4.4 : An i-transition θ is (t, N), where t is a transition (name) and $N = \{(t', n') \mid t' \in \mathit{conflict}^+(t)$, n' is the number of firings of the transition $t'\}$. □

From the above definition, it is rather easy to see the following statements hold.

1. An i-transition fires at most once in a run.
2. If any i-transition that fires in a run ρ fires also in another run α, then the firing order of transitions that fire in ρ is the same as that of α.

To prove **P1**, we define a notion *present* and a relation $\leq$ between two runs. Let $\mathit{prefix}_k(\rho)$ denote the prefix of length k of ρ.

Definition 4.5 : In the last state of $\mathit{prefix}_k(\rho)$, $\theta = (t, N)$ is said to be *present*, if for all $(t_i, n_i) \in N$, t_i fires exactly n_i times in $\mathit{prefix}_k(\rho)$. □

This presentness is precisely defined on the prefix of a run. However, for simplicity, we sometimes say assuming a particular run ρ that θ is present in a state of ρ. Let *present* $(M, s) = \{\theta \mid \theta = (t, N)$, $t \in M$, θ is present at $s\}$, where M is a set of transitions and s is a state.

Definition 4.6 : Let α and ρ be a run. $\rho \leq \alpha$, if any i-transition that fires in ρ fires also in α. $\Box$

Then, we can prove the following three lemmas.

Lemma 4.1 : For any just run α of N, there exists a run ρ extracted from $G(N, \varphi)$ such that $\rho \leq \alpha$.
(Sketch of proof) The proof is by induction on the length k of the prefix of ρ. The case $k = 1$ is trivial. Suppose that $\mathit{prefix}_k(\rho) \leq \alpha$. We show that at least one i-transition in $\mathit{present}\ (\mathit{firable}(s_k), s_k)$ fires also in α, where s_k is the last state of $\mathit{prefix}_k(\rho)$. Let $\theta_1 = (t_1, N_1) \in \mathit{present}\ (\mathit{firable}(s_k), s_k)$. If θ_1 fires in α, then we are done. Otherwise, since all i-transitions in $\mathit{prefix}_k(\rho)$ fires in α, some i-transition, say $\theta_2 = (t_2, N_2)$, corresponding to $\mathit{conflict}^+(t_1)$ must fire in α. If θ_2 is present and enabled at s_k, then t_2 must be in $\mathit{firable}(s_k)$. So, we are done. Otherwise, since again all i-transitions in $\mathit{prefix}_k(\rho)$ fires in α, some i-transition, say $\theta_3 = (t_3, N_3)$, corresponding to $\mathit{active}(t_2, s_k)$ must fire in α. We can consider that θ_3 to be a new θ_2, and repeat the above consideration. Since θ_3 fires before θ_2 in α, we can finally find an i-transition $\theta = (t, N)$ such that θ fires in α, and that θ is present and enabled at s_k. So, we are done. $\Box$

Lemma 4.2 : Let α be any just run of N. If ρ be a run extracted from $G(N, \varphi)$ such that $\rho \leq \alpha$, then ρ is just.
(Sketch of proof) If ρ is finite, then ρ is just. Otherwise, suppose a transition t is continuously enabled from s_a in ρ. Since the number of states are finite, there exists a root s of a SCC such that s appears in ρ infinitely often, and that t is enabled in the SCC. From the requirement 4 of Definition 4.3, for some j, t will eventually be included in $\mathit{firable}^j(s)$. Since $\rho \leq \alpha$ and α is just, t will fire or be disabled. This contradicts the hypothesis. $\Box$

Lemma 4.3 : Let α be any just run of N. If ρ be a run extracted from $G(N, \varphi)$ such that $\rho \leq \alpha$, then for all θ that fires in α, θ fires also in ρ.
(Sketch of proof) Suppose that there exists an i-transition that fires in α, but not fire in ρ. Without loss of generality, we can assume that θ is the first such i-transition in α. Suppose that the firings of θ_1, θ_2, $\cdots$, θ_n enable θ in α. From the hypothesis, θ_1, θ_2, $\cdots$, θ_n fire also in ρ. From this and $\rho \leq \alpha$, θ is enabled in ρ. Since ρ is just from Lemma 4.2 and again $\rho \leq \alpha$, θ will eventually fire in ρ. $\Box$

From Lemma 4.1 and Lemma 4.3, we have proven **P1**, hence, the correctness of our method.

5 Experimental results

In order to demonstrate the efficiency of our method, we show some experimental results in this section. As examples, we use nets that model some control parts of an asynchronous processor[UHN92]. Table 1 shows the sizes of those nets.

Table 1: Size of example nets.

	N_1	N_2	N_3	N_4
# of places	104	104	242	253
# of transitions	113	115	179	188

For the model checking, we also use very simple LTL formulas like :

$$\neg\Box(m\_req \rightarrow (m\_req\,\mathcal{U}\,reg\_data))$$

We don't mention the detail of the verification, but show only the performance of the partial order model checking comparing with that of the total order model checking. Table 2 shows the number of atoms generated for the model checking and CPU times for each net. The program is coded in C language. The CPU times were measured on a 25 Mips workstation. N_1 and N_2 seem to have less concurrency. Thus, although the reduced state (atom) space is smaller than the full state (atom) space, the partial order model checking is inefficient due to its overhead. On the other hand, N_3 and N_4 seem to have much concurrency. Therefore, the partial order model checking is much more efficient than the total order model checking. One clear advantage of our approach is that it is fully automatic, while the disadvantage is that the state space generation is necessary for each formula.

6 Conclusion

Mainly for the verification of asynchronous circuits, we have developed the concrete algorithm for the partial order model checking of one-safe Petri nets and a linear-time temporal logic without the next-time operator. The partial order model checking basically consists of the reduced state space generation and the conventional LTL model checking. The reduced state space generation is based on the fact that the firings of the transitions are

Table 2: Performance of the model checking algorithms.

Total order MC	N_1	N_2	N_3	N_4
# of atoms	3689	11593	19941	($>$ 700000)
CPU time (sec)	9.3	29.3	103.3	($>$ 145 min)

(Total order MC could not finish N_4 by lack of memory)

Partial order MC	N_1	N_2	N_3	N_4
# of atoms	2255	7079	563	17424
CPU time (sec)	11.8	39.1	4.8	174.5

only partially ordered by causality and a given formula, and that the order of firings of transitions not related by this partial order is irrelevant for the evaluation of the given formula. Further, we handle only just runs, because (1) it is suitable to model asynchronous circuits, and (2) it makes the method efficient. Unfortunately, The partial order model checking does not improve the worst case time complexity, since it is possible that there is no concurrency in the given net, in which case the reduced state space may be the same as the full state space. However, the experimental results demonstrate that the partial order model checking is sometimes much more efficient than the conventional (total order) model checking.

We are now trying to verify control parts of an asynchronous processor using this presented method. This method can also be extended very easily to the partial order model checking for time Petri nets and a real-time logic, which is suitable for the verification of real-time systems. This is another research project that we are devoted to.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and Analysis of Computer Algorithms.* Addison-Wesly, 1974.

[BCD+90] J. R. Burch, E. M. Clarke, D. L. Dill, L. J. Hwang, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Proc. of 5th IEEE LICS*, 1990.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal-logic specifications. *ACM Trnas. on Programming Languages and Systems*, 8(2):244–263, 1986.

[DNS90] D. L. Dill, S. M. Nowick, and R. F. Sproull. Specification and automatic verification of self-timed queues. In *Formal verification of hardware design*. IEEE computer society press, 1990.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. *Proc. of Computer Aided Verification Workshop*, 1990.

[GW91] P. Godefroid and P. Wolper. A partial approach to model checking (extended abstract). *??*, 1991.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finaite state concurrent programs satisfy their linear specification. *Proc. of 12nd POPL*, 1985.

[McM92] K. L. McMillan. *Symbolic model checking : An approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[UHN92] Y. Ueno, I. Honma, and T. Nanya. Note on 2-phase asynchronous processor organization. *Spring Convention of IPSJ*, 1992.

[Val90] A. Valmari. A stubborn attack on state explosion. *Proc. of Workshop on Comupter-Aided Verification*, 1990.

[YTK91] T. Yoneda, Y. Tohma, and Y. Kondo. Acceleration of timing verification method based on time petri nets. *Systems and Computers in Japan*, 22(12):37–52, 1991.