Level Oriented Formal Model for Asynchronous Circuit Verification and its Efficient Analysis Method *

Tomoya Kitai, Yusuke Oguro
Tokyo Institute of Technology
{kitai, yoguro}@yt.cs.titech.ac.jp

Tomohiro Yoneda
National Institute of Informatics
yoneda@nii.ac.jp

Eric Mercer
Brigham Young University
egm@cs.byu.edu

Chris Myers
University of Utah
myers@vlsigroup.ece.utah.edu

Abstract

Using a level-oriented model for verification of asynchronous circuits helps users to easily construct formal models with high readability or to naturally model data-path circuits. On the other hand, in order to use such a model on large circuits, techniques to avoid the state explosion problem must be developed. This paper first introduces a level-oriented formal model based on time Petri nets, and then proposes its partial order reduction algorithm that prunes unnecessary state generation while guaranteeing the correctness of the verification.

Key words *Level-oriented model, timed asynchronous circuits, formal verification, time Petri nets.*

1 Introduction

Many formal verification algorithms for asynchronous circuits that are based on the exploration of reachable states use *transition-oriented* models such as Petri nets and CSP in order to model circuits and specifications [1, 2, 3, 4, 5]. In this approach, the behavior of an asynchronous circuit is represented using *transitions of signals*. This representation has the potential ability to model the real nature of asynchronous control circuits. It is, however, not easy for nonexpert users to construct good and comprehensive representations on this model. Furthermore, in asynchronous circuit design, control signals are sometimes embedded in data-path circuits. An example of this is a dual-rail encoding, which requires some (abstracted) data-path circuits to be formally modeled for verification. In this type of application, a transition-oriented model is not suitable.

This paper tries to represent the behavior of asynchronous circuits also using *values of signals* like those used in the synchronous circuit design process. For this purpose, a *level-oriented* model is first introduced. Our model, which we call LTN (Level Time Petri Net), is obtained by extending time Petri nets such that firing an LTN transition can assign values to a set of boolean variables and that the validity of an expression over the boolean variables is also used as an enabling condition of an LTN transition in addition to the marking. Thus, an LTN can easily model the behavior based on both changes of signals and values of signals.

On the other hand, an approach to analyzing this new model in a traditional total order manner is not acceptable for large circuits due to state explosion. In other words, a new model is useless without an efficient analysis algorithm. For transition-oriented models, two major methods are proposed for this purpose, implicit state space enumeration based on BDDs [3] and partial order reduction [4, 5]. Since our current interest is in verifying timed circuits with bounded delays and the implicit state representation method often fails to efficiently represent timed states (in particular, sets of inequalities), this work chooses the partial order reduction approach.

Timed automata [6] can also be used as a level oriented model, and partial order reduction has been applied to their analysis [7, 8]. Our experience, however, has found that the generality of timed automata comes at an increase in analysis complexity, and this increased generality does not appear to be necessary for verifying asynchronous circuits.

Several alternative level-oriented Petri net models have been proposed such as TEL structures [9], level-ruled Petri nets (LPNs) [10], and an extension of time Petri nets [11]. An LTN is obtained by refining the one proposed in [11]. An LTN is somewhat less expressive than TEL structures and LPNs. In particular, timing annotations and Boolean conditions are placed on the transition in an LTN while they are placed on the edge between the place and transition in TEL structures and LPNs. This increased expressiveness, however, comes at a cost in the analysis algorithm's complexity. As a result, the algorithms for analysis of these nets have tended to be conservative rather than exact [9, 10]. To the best of our knowledge, the work in [10, 12] is the

*This research is supported by NSF Japan Program award INT-0087281, SRC contract 99-TJ-694, a grant from Intel Corporation, and JSPS Joint Research Projects.

Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)
0-7695-1852-4/02 $17.00 © 2002 IEEE

only one that proposes a partial order reduction for a level-oriented Petri net model, namely the LPN model, though the algorithm is conservative. The goal of this work is to obtain the exact verification results using the LTN model.

This paper is organized as follows. The next section introduces the LTN model. Section 3 briefly reviews the verification method used in this paper. Section 4 proposes the partial order reduction algorithm for an LTN. Section 5 shows the experimental results obtained by verifying several examples with the proposed algorithm. Finally, we summarize our results in Section 6.

2 Level Oriented Model

A traditional time Petri net consists of *transitions* (thick bars), *places* (circles), and *arcs* between transitions and places. A *token* (large dot) can occupy a place, and when every source place of a transition is occupied, the transition becomes *enabled*. Each transition has two times, the *earliest firing time* and the *latest firing time*. An enabled transition becomes ready to fire (i.e., *firable*) when it has been continuously enabled for its earliest firing time, and cannot be continuously enabled for more than the latest firing time, i.e., it must fire unless it is disabled. The firing of a transition occurs instantly. It consumes tokens in its source places and produces tokens into its destination places.

In an LTN, two additional functions assign and condition can be associated with a transition. The assign function relates a transition to assignments on Boolean variables, and the condition function relates a transition to an expression over boolean variables. The enabling condition of an LTN is extended, and a transition is *enabled* if both the expression given by condition is true and every source place is occupied. For example, in the LTN shown in Figure 1(a), t_c is enabled only if $b_1 \wedge b_2$ is true. The firing rule of an LTN is also extended in that when a transition fires, the assignments specified by the assign function are done while consuming and producing tokens. For example, in an LTN shown in Figure 1(b), when t_a fires, a_1 and a_2 are set to 1 and 0, respectively. Using assign and condition, a level-oriented model can easily be described.

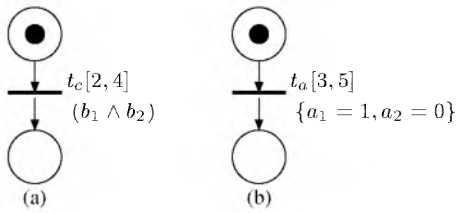

Figure 1. An example of an LTN

2.1 Formal Definitions of LTN

An LTN N is a ten-tuple, $N = (P, T, F, \mathsf{Eft}, \mathsf{Lft}, V, \mathsf{assign}, \mathsf{condition}, \mu^0, val^0)$. The members P, T, F, Eft, Lft, and μ^0 are the same as those of the time Petri net; although the members V, assign, condition, and val^0 are newly added for an LTN, and defined as follows:

- V is a finite set of Boolean variables.
- $\mathsf{assign} : T \to 2^{\mathcal{A}}$, where $\mathcal{A} = \{v = b | v \in V, b \in \{0, 1\}\}$.
- $\mathsf{condition} : T \to \mathcal{C}$, where $\mathcal{C} = \{f_1 \vee f_2 | f_1, f_2 \in \mathcal{C} \cup \mathcal{F}\}$ and $\mathcal{F} = \{g_1 \wedge g_2 | g_1, g_2 \in \mathcal{F} \cup V \cup \overline{V}\}$. $\overline{V}$ denotes $\{\overline{v_0}, \overline{v_1}, \overline{v_2}, \ldots\}$ if $V = \{v_0, v_1, v_2, \ldots\}$.
- $val^0 : V \to \{0, 1\}$ is for the initial values of Boolean variables.

The assign function relates a transition to assignments on Boolean variables performed on the firing of the transition. For example, $\mathsf{assign}(t) = \{a = 1, b = 0\}$. The condition function specifies a Boolean expression that should be true for the transition to be enabled. This expression is represented by a sum-of-products such as,

$$\mathsf{condition}(t) = a\,b\,\overline{c} \vee d\,\overline{e},$$

where a, b, c, d, e are Boolean variables ($\wedge$ are omitted here).

A state of an LTN is a tuple (μ, I, val), where μ is a marking, I is a set of inequalities, and val is an assignment of a value to each Boolean variable. For a transition t, two kinds of timing variables, a *past variable* and a *future variable* are used. A past variable represents its most recent firing time, and a future variable represents its next firing time. This paper uses t also for the past variable, and $\ddot{t}$ for the future variable. Inequalities in I are over these variables. For a Boolean expression f and an assignment val, $\mathsf{eval}(f, val)$ denotes the value of f under val. Thus, a set of enabled transition in a state $s = (\mu, I, val)$ can be expressed as

$$\mathsf{enabled}(\mu, val) = \{t \mid {\bullet t} \subseteq \mu, \mathsf{eval}(\mathsf{condition}(t), val) = 1\},$$

where ${\bullet t}$ denotes the set of source places of t. A transition is *firable*, if it is enabled and possible to fire earlier than any other enabled transitions. That is,

$$\mathsf{firable}(s) = \{t \mid (I \cup \{\ddot{t} \leq \ddot{t'} \mid t' \in \mathsf{enabled}(\mu, val)\}) \text{ is consistent}\}.$$

In this work, only 1-safe LTNs are considered, i.e., in any reachable state $s = (\mu, I, val)$, no transition t such that $(\mu - {\bullet t}) \cap t{\bullet} \neq \emptyset$ is firable. Similarly, it is assumed that no transition has vacuous assignment, i.e., for any variable v, and in any reachable state $s = (\mu, I, val)$ with $val(v) = b$, no transition t such that "$v = b$" $\in \mathsf{assign}(t)$ is firable. These assumptions are just for simplification of our algorithm, and with an increase in complexity they can be removed.

Figure 2(a) shows a NOR gate model with a hazard detection mechanism represented by a time Petri net. The

Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)
0-7695-1852-4/02 $17.00 © 2002 IEEE

marking shown in this figure represents the state with input $a = 0, b = 0$ and output $c = 1$. If a or b goes high, $c-$ is enabled. However, after a goes high, $a-$ is not enabled until c goes low. Therefore, a hazard caused by $a+$ and $a-$ before $c-$ is detected as a failure (as described in the next section). Figure 2(b) shows the corresponding NOR gate model by a LTN. In this model, the enabling conditions are straightforwardly represented by condition functions. When a hazard occurs, a dummy transition $err+$ is enabled, and it immediately fires resulting in a failure, because the corresponding input transition is always disabled. It can be seen that an LTN represents a model more concisely than a time Petri net.

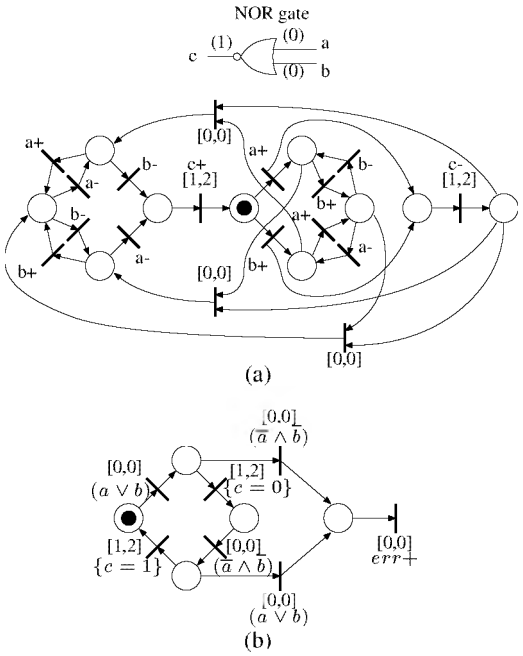

Figure 2. NOR gate models expressed by a time Petri net and an LTN.

3 Verification Method

This paper uses the *timed trace theoretic verification* method [13]. A *module* is a tuple (I, O, N), where I and O are sets of input and output wires respectively, and N is an LTN. We use a module as a formal model for a circuit element (e.g., a gate) and a specification. Some transitions in N correspond to wires, and the firing of those transitions change the values of the wires. A transition related to an input wire of the module is called an *input transition*. An *output transition* is defined similarly. Moreover, the Boolean variables of an LTN correspond to input or output wires. A circuit consists of a set of modules. In a set of modules, input transitions fire only in synchronization with the corresponding output transition with the same wire name in some different module. When an output transition fires, if no input transitions are enabled in a module, a *failure* occurs. This represents that a module tries to send an output but some other module cannot receive it as a corresponding input. Thus, it is the case that some bad output can be produced. In this sense, our verification method checks safety properties. Note that Boolean variables can be changed at any time without failures.

We define the following, where $s = (\mu, I, val)$:

- $\mathsf{out\_trans}(t)$ is the output transition that corresponds to t. If t is an output transition, then $\mathsf{out\_trans}(t)$ is t itself.
- $\mathsf{in\_trans}(t)$ is a set of input transitions that correspond to $\mathsf{out\_trans}(t)$.
- $\mathsf{sync\_trans}(s, t) = \{\mathsf{out\_trans}(t)\} \cup \big(\mathsf{in\_trans}(t) \cap \mathsf{enabled}(\mu, val)\big)$.

4 Verification Algorithm

The following shows a skeleton of the verification algorithm based on the partial order reduction.

```
1:  verify(s)
2:    begin
3:      if (s is already visited) then return(true);
4:      if (s is a failure state) then return(false);
5:      Mark s as visited;
6:      forall (t_f ∈ ready(s));
7:        forall (s' ∈ successo (s, t_f));
8:          if (verify(s') is false) then return(false);
9:      return(true);
10:   end
```

Although this algorithm is quite similar to the usual depth-first search algorithm, there are some major differences that characterize the partial order reduction. One is that $\mathsf{ready}(t)$ is the subset of firable transitions, and the other is that multiple states are generated at the firing of t_f by successo (s, t_f). We show how to construct $\mathsf{ready}(s)$ and successo (s, t_f) for an LTN in the next subsections.

4.1 $\mathsf{ready}(s)$

$\mathsf{ready}(s)$ is the set of output transitions firable and necessary to fire in s in order to determine if a circuit is correct. For a firable output transition t and a state s, if $\mathsf{dependent}(s, t)$ denotes the set of output transitions (including t) such that the interleaving of the firings of those transitions should be considered, $\mathsf{ready}(s)$ is defined as

$$\mathsf{ready}(s) = \begin{cases} \mathsf{minset}(\pi_1, \cdots, \pi_m) & \text{if } m > 0 \\ \mathsf{firable}(s) & \text{otherwise} \end{cases}$$

where

$$\{\pi_1, \cdots, \pi_m\} = \{\mathsf{dependent}(s, t) \mid t \in \mathsf{firable}(s), \mathsf{dependent}(s, t) \subseteq \mathsf{firable}(s)\}.$$

Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)
0-7695-1852-4/02 $17.00 © 2002 IEEE

minset$(\pi_1, \pi_2, \cdots)$ chooses the set with smallest cardinality from $\pi_1, \pi_2, \cdots$. Since dependent(s, t) may include transitions which are not firable, those dependent sets are not chosen.

dependent(s, t_f) is the smallest set which satisfies the following:

1. $t_f \in$ dependent(s, t_f).
2. If $t \in$ dependent(s, t_f), then
   (a) $\forall t_x \in \bigcup_{t' \in \text{sync\_trans}(s,t)}$ conflict(t').[active$(s, t,$ necessa ry$(s, t_x, \{t\})) \subseteq$ dependent(s, t_f)].
   (b) $\forall t_x \in \bigcup_{t' \in \text{sync\_trans}(s,t)}$ hide_fail(t').[active$(s, t,$ necessa ry$(s, t_x, \{t\})) \subseteq$ dependent(s, t_f)].
   (c) $\forall t_x \in \bigcup_{t' \in \text{sync\_trans}(s,t)}$ ac_conf(s, t').[active$(s, t,$ necessa ry$(s, t_x, \{t\})) \subseteq$ dependent(s, t_f)].
   (d) $\forall t_x \in \{t' \mid t' \in$ sync_trans$(s, t),$ condition$(t') \neq \emptyset\}$.[active$(s, t,$ ac_necessary$(s,$ condition$(t_x), 0, \{t\})) \subseteq$ dependent(s, t_f)].
   (e) $\forall t_x \in \bigcup_{t' \in \text{sync\_trans}(s,t)}$ affected(t').[active$(s, t,$ ac_dependent$(s,$ condition$(t_x), t)) \subseteq$ dependent(s, t_f)].

As mentioned later, necessary, ac_necessary and ac_dependent contain sets of pairs (u, τ), where u is a transition and τ is the minimal time necessary to fire u. Thus, if τ is large enough, the algorithm does not have to consider the firing order of t_f and u because u is enabled too late. Those transitions which fire too late are omitted from the sets by active. In other words, active(s, t, TT) represents a set of transitions u such that $(u, \tau) \in TT$ and u can fire τ time units earlier than t. Therefore, dependent(s, t_f) includes only those active transitions.

Now, we explain each of the above conditions using examples. Conditions 2.(a) and 2.(b) are the same as those for the transition-oriented model, so we omit them here. The details can be found in [14]. Condition 2.(c) states that the firing order of t and t_x should be considered, if t is in the dependent set and t makes the condition of t_x false. In this condition,

$$\text{ac\_conf}(s, t) = \{t_c \mid \text{eval}(\text{condition}(t_c), val) = 1, \text{eval}(\text{condition}(t_c), \text{newval}(s, t)) = 0\},$$

where newval(s, t) represents the assignment of variables after t fires. Consider the case shown in Figure 3(a). If t and t_1 are fired only in this order and the initial value of a is 0, the firing of t_2 is missed. However, t_2 can actually fire, if t_1 and t_x fire earlier than t. If the firing of t_2 causes a failure, omitting Condition 2.(c) implies that the algorithm misses a failure that may actually occur. Thus, if t is in the dependent set, the algorithm must obtain t_1 by using necessary$(s, t_x, \{t\})$ so that the chance that t_x fires earlier than t is covered.

necessary(s, t, T_D) contains the set of pairs (u, τ), where u is an output transition enabled in s which must fire in order to fire t under the condition that transitions in T_D are not fired, and τ is the minimal time difference between the firings of u and t. necessary(s, t, T_D) is defined as follows. Note that $t_{out} =$ out_trans(t), because if t is an input transition, its corresponding output transition should be considered.

1. If $t_{out} \in T_D$, then necessary$(s, t, T_D) = \emptyset$.
2. If $t_{out} \in$ enabled(μ, val), then necessary$(s, t, T_D) = \{(t_{out}, 0)\}$.
3. Otherwise, necessary$(s, t, T_D) = \{(t_1, \text{Eft}(t_{out}) + \tau_1), \cdots, (t_l, \text{Eft}(t_{out}) + \tau_l)\}$, where
   - $\{(t_1, \tau_1), \cdots, (t_l, \tau_l)\} =$ minset$(\pi_1, \cdots, \pi_j)$.
   - $\{\pi_1, \cdots, \pi_j\} = \{\bigcup_{t' \in \bullet p}$ necessa ry$(s, t', T_D \cup \{t_{out}\}) \mid p \in \bullet t_{out} - \mu\} \cup \{\pi' \mid \pi' =$ ac_necessa ry$(s,$ condition$(t_{out}), 1, T_D \cup \{t_{out}\}), \pi' \neq \emptyset\}$

ac_necessary(s, f, b, T_D) also contains a set of pairs (u, τ). The difference from necessa ry is that u is the transition enabled in $s = (\mu, I, val)$, of which firing is necessary to let the expression f take value b. ac_necessa ry(s, f, b, T_D) is defined as follows, where assign_trans$(v, b) = \{t \mid$ "$v = b$" $\in$ assign$(t)\}$:

1. If $b =$ eval(f, val), then ac_necessary$(s, f, b, T_D) = \emptyset$.
2. If f is a positive form of variable v, then ac_necessary$(s, f, b, T_D) = \bigcup_{t' \in \text{assign_trans}(v,b)}$ necessary(s, t', T_D).
3. If f is the negative form of variable v, then ac_necessary$(s, f, b, T_D) = \bigcup_{t' \in \text{assign_trans}(v,\bar{b})}$ necessary(s, t', T_D).
4. If f is $f_1 \wedge f_2 \wedge \cdots \wedge f_n$ with $b = 1$, or $f_1 \vee f_2 \vee \cdots \vee f_n$ with $b = 0$, then ac_necessary$(s, f, b, T_D) =$ minset$(\pi_1, \cdots, \pi_m)$, where $\{\pi_1, \cdots, \pi_m\} = \{\pi'_i \mid 1 \leq i \leq n, \pi'_i =$ ac_necessary$(s, f_i, b, T_D), \pi'_i \neq \emptyset\}$
5. If f is $f_1 \wedge f_2 \wedge \cdots \wedge f_n$ with $b = 0$, or $f_1 \vee f_2 \vee \cdots \vee f_n$ with $b = 1$, then ac_necessary$(s, f, b, T_D) = \bigcup_{i=1,n}$ ac_necessary(s, f_i, b, T_D).

On the other hand, for a transition t in the dependent set, Condition 2.(d) of dependent(s, t_f) checks the fireability of transitions that make the condition of t false. For example,

Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)
0-7695-1852-4/02 $17.00 © 2002 IEEE

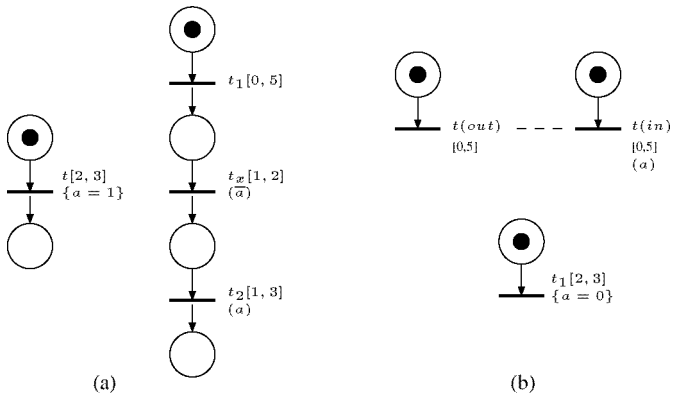

Figure 3. Examples of 2.(c) and (d)

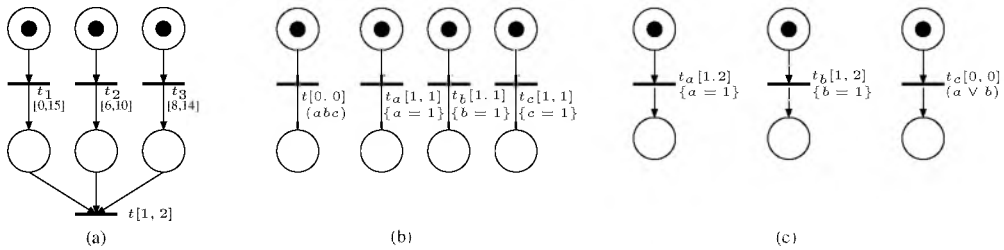

Figure 4. Examples for handling true parents

in Figure 3(b), if the output transition t(out) fires, the corresponding input transition t(in) also fires synchronously, and no failure occurs. However, if t_1 fires earlier than t(in), then t(in) is disabled, and a failure occurs when t(out) fires. If t(out) is always fired earlier than t_1, this failure is missed. Thus, we need Condition 2.(d) in order to fire transitions which make conditions of other transitions false. This is done by ac_necessa ry

Conditions 2.(e) is for making it easier to decide *true parents* of newly enabled transitions. A true parent of an enabled transition t is a transition that actually makes t enabled and hence decides its firing time. For example, since t has multiple source places in Figure 4(a), the transition that produces the final token to the source places of t can be a true parent. Moreover, if condition(t) is a simple product as shown in Figure 4(b), the transition that assigned true to a Boolean variable last can be a true parent. Note that even if t_3 fires last in a firing sequence that enables t in Figure 4(a), it does not mean that only t_3 is a true parent of t in that firing sequence. This is because the partial order reduction algorithm does not usually give the ordering relation among concurrent transitions such as t_1, t_2, and t_3. Thus, when t becomes enabled, the possibilities that each of t_1, t_2 and t_3 is a true parent of t is checked, and a new state is generated by giving timing constraints for such a transition to be a true parent. However, if condition(t) contains logical OR operators, true parents should be decided in a different way. For example, in Figure 4(c), either a transition t_a or t_b that fires earlier than the other can be a true parent of t_c, and the firing of such a transition immediately makes t enabled. Therefore, the decision of true parents cannot be postponed until all candidates of true parents fire. Hence, in the case where condition(t_c) is a sum of product terms, when one of the product terms becomes true by firing t, we check whether other product terms of condition(t_c) can be true by firing transitions t', and give the ordering relation between t and t'. This implies that all of possible true parent candidates of t_c are explicitly ordered, and it allows us to decide true parents of this type according to the firing sequences.

In Condition 2.(e),

$$\text{affected}(t) = \{t_c \mid \text{var}(\text{assign}(t)) \cap \text{var}(\text{condition}(t_c)) \neq \emptyset\}$$

and

$$\text{var}(f) = \{v \mid v \text{ is a variable included in } f\}.$$

By using ac_dependent(s, f, t), which is defined below, Condition 2.(e) checks the fireability of the transitions that makes the other products true.

1. If $f = f_1 \vee f_2 \vee \cdots \vee f_n$ and eval$(f, val) = 0$ and eval$(f, \text{newval}(s, t)) = 1$, then

Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)
0-7695-1852-4/02 $17.00 © 2002 IEEE

$$\mathsf{ac\_dependent}(s, f, t) = \bigcup_{i=1,n} \mathsf{ac\_necessary}(s, f_i, 1, \{t\})$$

2. Otherwise,
   $\mathsf{ac\_dependent}(s, f, t) = \emptyset$.

4.2 successo (s, t_f)

When we fire t_f such that $t_f \in \mathsf{ready}(s)$ in $s = (\mu, I, val)$, the following processes are needed.

- For each $u \in \mathsf{ready}(s)$, the constraint $t_f \leq \ddot{u}$ is added to I, where $\ddot{u}$ is the future variable of u.
- For a transition t_n newly enabled by firing of t_f, its true parent is decided, and the appropriate constraints for it are added to I.

We can consider two types of true parents for t_n, the transitions that produce tokens in source places of t_n, and the transitions that satisfy $\mathsf{condition}(t_n)$. The former are called *place-related true parents*, and the latter are called *condition-related true parents*. In order to decide true parents of t_n, $\mathsf{true\_parent}(s, t_n)$, which is actually defined in 4.2.1, denotes a set of pairs (t_p, I_p), where t_p is a true parent of t_n, and I_p is a set of inequalities that are necessary for t_p to be a true parent. Note that if adding I_p to I of the current state makes I inconsistent, such (t_p, I_p) is discarded during the state generation process.

When a true parent t_p of newly enabled transition t_n is decided, the following constraint

$$\mathsf{Eft}(t_n) \leq \ddot{t}_n - t_p \leq \mathsf{Lft}(t_n)$$

is added to I. In this case, if t_p does not become a true parent of any other transitions, t_p can be removed from I by $\mathsf{delete}(I, D)$. $\mathsf{delete}(I, D)$ removes from I the inequalities including variables in the set D without affecting the solution set projected to the remaining variables. Since the behavior of the transition t_n can be represented by a future variable $\ddot{t}_n$, failures are not missed by removing t_p. Furthermore, if t_p is not removed, the state enumeration process may not terminate.

On the other hand, for a transition t which is not enabled in s, it is possible that I contains the past variables of transitions that can be true parents of t when it becomes enabled, e.g., the transitions which produce tokens in the source places of t. Such past variables must not be removed in general. $\mathsf{need\_to\_keep}(s, t)$ shown in 4.2.3 denotes the set of such transitions. Using this, the set of variables that can be removed is defined as $T - \{t \mid \exists t' \in T - \mathsf{enabled}(\mu, val).[t \in \mathsf{need\_to\_keep}(s, t')]\}$, where T is the set of all output transitions.

¿From these definitions, $\mathsf{successo}\ (s, t_f)$ consists of states $s' = (\mu', I', val')$ that satisfy the following conditions.

1. $\mu'' = \mu - {}^\bullet t_f$.
2. $\mu' = \mu'' \cup t_f{}^\bullet$.
3. $val' = \mathsf{newval}(s, t_f)$.
4. Obtain J_0 from I by replacing the future variable $\ddot{t}_f$ with the past variable t_f.
5. $J_1 = J_0 \cup \{t_f \leq \ddot{u} \mid u \in \mathsf{ready}(s)\}$.
6. $J_2 = \mathsf{delete}(J_1, \{\ddot{t} \mid t \in \mathsf{enabled}(\mu, val) - \mathsf{enabled}(\mu'', val')\})$.
7. For the set of newly enabled transitions by firing t_f, $E = \mathsf{enabled}(\mu', val') - \mathsf{enabled}(\mu'', val) = \{t_{n_1}, \cdots, t_{n_l}\}$, the true parent assignment, $\{t_{p_1}, \cdots, t_{p_l}\}$ such that $(t_{p_k}, I_{p_k}) \in \mathsf{true\_parent}(s, t_{n_k})$ for $1 \leq k \leq l$ is valid if $J_2 \cup \bigcup_{k=1,l} I_{p_k}$ is consistent, and in this case obtain J_3 as follows.

$$J_3 = J_2 \cup \bigcup_{k=1,l} [I_{p_k} \cup \{\mathsf{Eft}(t_{n_k}) \leq \ddot{t}_{n_k} - t_{p_k} \leq \mathsf{Lft}(t_{n_k})\}]$$

8. For $s'' = (\mu', J_3, val')$ and $D = T - \{t \mid \exists t' \in T - \mathsf{enabled}(\mu', val').[t \in \mathsf{need\_to\_keep}(s'', t')]\}$, let $I' = \mathsf{delete}(J_3, D)$.

In 7., since more than one true parent assignment, $\{t_{p_1}, \cdots, t_{p_l}\}$, can exist, $\mathsf{successo}\ (s, t_f)$ can contain more than one state. We define $\mathsf{true\_pa\ rent}(s, t_n)$ and $\mathsf{need\_to\_keep}(s, t)$ in the following subsections.

4.2.1 $\mathsf{true\_pa\ rent}(s, t_n)$

Let $\mathsf{ruler}(t_n, I) = {}^\bullet{}^\bullet t_n \cap \mathsf{var}(I)$ denote a set of candidates of place-related true parents for t_n, and a set of the pairs of place-related true parents and their necessary constraints is denoted by

$$\mathsf{true\_parent\_place}(s, t_n) = \bigcup_{t_p \in \mathsf{ruler}(t_n, I)} \{(t_p, I_p) \mid I_p = \{t' \leq t_p \mid t' \in \mathsf{ruler}(t_n, I)\}\}.$$

Furthermore, for condition-related true parents, $\mathsf{true\_parent\_cond}(s, f)$ denotes a set of pairs (t_p, I_p), similarly. The definition of $\mathsf{true\_parent\_cond}(s, f)$ is given in the next subsection.

¿From these, for cases where a place-related true parent becomes the actual true parent, $t'_p \leq t_p$ is necessary for each $(t_p, I_p) \in \mathsf{true\_pa\ rent\_place}(s, t_n)$ and $(t'_p, I'_p) \in \mathsf{true\_parent\_cond}(s, \mathsf{condition}(t_n))$, and

$$TP_1 = \{\ (t_p, \{t'_p \leq t_p\} \cup I_p \cup I'_p)\ \mid\ (t_p, I_p) \in \mathsf{true\_parent\_place}(s, t_n),\ (t'_p, I'_p) \in \mathsf{true\_parent\_cond}(s, \mathsf{condition}(t_n))\ \}$$

is obtained. Similarly, for cases where a condition-related true parent becomes the actual true parent, $t_p \leq t'_p$ is necessary, and

$$TP_2 = \{\ (t'_p, \{t_p \leq t'_p\} \cup I_p \cup I'_p)\ \mid\ (t_p, I_p) \in \mathsf{true\_parent\_place}(s, t_n),\ (t'_p, I'_p) \in \mathsf{true\_parent\_cond}(s, \mathsf{condition}(t_n))\ \}$$

Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)
0-7695-1852-4/02 $17.00 © 2002 IEEE

is obtained. Therefore, true_parent(s, t_n) can be defined as follows:

1. If condition$(t_n) = \emptyset$, only place-related true parents exist, and hence
   true_parent$(s, t_n) =$ true_pa rent_place(s, t_n)
   holds.
2. Otherwise, by considering both cases,
   true_parent$(s, t_n) = TP_1 \cup TP_2$
   holds.

4.2.2 true_parent_cond(s, f)

The candidates of true parent for the condition of the form "$v = b$" is denoted by vruler$(v, b, I) =$ assign_trans$(v, b) \cap$ var(I). A set of pairs of the condition-related true parents for f and their necessary constraints is obtained as follows:

1. If f is a positive form of variable v, then
   true_parent_cond$(s, f) =$
   $\{(t_p, \emptyset) \mid t_p \in$ vruler$(v, 1, I)\}$.
2. If f is the negative form of variable v, then
   true_parent_cond$(s, f) =$
   $\{(t_p, \emptyset) \mid t_p \in$ vruler$(v, 0, I)\}$.
3. If f is $f_1 \wedge f_2$, then
   true_pa rent_cond$(s, f) =$
   $\{(t_p, \{t'_p \le t_p\} \cup I_p \cup I'_p) \mid$
   $(t_p, I_p) \in TT_1, (t'_p, I'_p) \in TT_2\} \cup$
   $\{(t'_p, \{t_p \le t'_p\} \cup I_p \cup I'_p) \mid$
   $(t_p, I_p) \in TT_1, (t'_p, I'_p) \in TT_2\}$,
   where $TT_1 =$ true_pa rent_cond(s, f_1), and $TT_2 =$ true_parent_cond(s, f_2)
4. If f is $f_1 \vee f_2$, then
   - If eval$(f_1, val) = 1$ and eval$(f_2, val) = 0$, then true_pa rent_cond$(s, f) = TT_1$
   - If eval$(f_1, val) = 0$ and eval$(f_2, val) = 1$, then true_pa rent_cond$(s, f) = TT_2$
   - If eval$(f_1, val) = 1$ and eval$(f_2, val) = 1$, then
     true_pa rent_cond$(s, f) =$
     $\{(t_p, I_p \cup \{t_p \le t\}) \mid (t_p, I_p) \in TT_1, t \in T_2\} \cup$
     $\{(t'_p, I'_p \cup \{t'_p \le t\}) \mid (t'_p, I'_p) \in TT_2, t \in T_1\}$,

   where $TT_1 =$ true_parent_cond(s, f_1), $TT_2 =$ true_parent_cond(s, f_2), $T_1 = \{t_p \mid (t_p, I_p) \in TT_1\}$, and $T_2 = \{t'_p \mid (t'_p, I'_p) \in TT_2\}$.

Note that if f consists of more than two terms in (3) or (4), true_pa rent_cond(s, f) is applied recursively.

Consider the example shown in Figure 5. Suppose that the state s_1 is obtained by the firing sequence t_a, t_b, t_c, t_1. In this case, for $f = a \vee bc$, we have $TT_1 = \{(t_a, \emptyset)\}$ and $TT_2 = \{(t_b, \{t_b \ge t_c\}), (t_c, \{t_c \ge t_b\})\}$. Thus, true_pa rent_cond$(s_1, a \vee bc) = \{(t_a, \{t_a \le t_b\}), (t_a, \{t_a \le t_c\}), (t_b, \{t_b \ge t_c, t_b \le t_a\}), (t_c, \{t_c \ge t_b, t_c \le t_a\})\}$ holds.

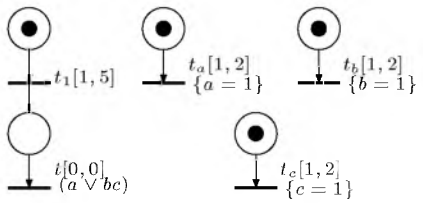

Figure 5. Example of true_parent_cond(s, f)

4.2.3 need_to_keep(s, t)

For a disabled transition t, need_to_keep(s, t) is a set of transitions that can be true parents of t. First, let

last$(T_p, I) =$
$\{t \mid t \in T_p, (\{t \ge t' \mid t' \in T_p\} \cup I)$ is consistent$\}$

denote the set of transitions which can fire later than any other transitions in T_p.

There are two cases to consider: first, a place-related true parent becomes the actual true parent; and second a condition-related true parent becomes the actual true parent. The set of place-related true parents is $TP_1 =$ last(ruler$(t, I), I)$, and the set of condition-related true parents is $TP_2 =$ need_cond$(s,$ condition$(t))$, where need_cond(s, f) is obtained as follows:

1. If f is the positive form of variable v, and
   - If eval$(v, val) = 1$, then need_cond$(s, f) =$ {vruler$(v, 1, I)$}
   - Otherwise, need_cond$(s, f) = \emptyset$
2. If f is the negative form of variable v, and
   - If eval$(\overline{v}, val) = 1$, then need_cond$(s, f) =$ {vruler$(v, 0, I)$}
   - Otherwise, need_cond$(s, f) = \emptyset$
3. If f is of the form $f_1 \wedge f_2 \wedge \cdots \wedge f_n$, then need_cond$(s, f) =$ last$(\bigcup_{i=1,n}$ need_cond$(s, f_i), I)$
4. If f is of the form $f_1 \vee f_2 \vee \cdots \vee f_n$, then need_cond$(s, f) = \bigcup_{i=1,n}$ need_cond(s, f_i)

The set of true parents can be obtained by applying last to the union of these two sets, that is, last$(TP_1 \cup TP_2, I)$. However, in order to make t enabled, the firings of other transitions t' are necessary, and if $t_p \in$ last$(TP_1 \cup TP_2, I)$ cannot fire later than those transitions, t_p cannot be the actual true parent. can_fire_last(s, t, t_p, T_D) checks this possibility.

Therefore, need_to_keep(s, t) can be defined as follows

need_to_keep$(s, t) = \{t_p \mid t_p \in$ last$(TP_1 \cup TP_2, I)$,
can_fire_last$(s, t, t_p, \emptyset)\}$.

Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)
0-7695-1852-4/02 $17.00 © 2002 IEEE

Table 1. Experimental results (1).

	Total				Partial			
No. of stages	6	7	8	9	6	7	8	9
No. of states	9096	16376	24784	37728	270	324	334	402
CPU times (sec)	13.8	26.7	48.3	84.0	0.05	0.08	0.04	0.09
Memory usage (MB)	17.2	37.4	68.5	116	0.28	0.32	0.33	0.38

Table 2. Experimental results (2).

	Proposed				VINAS-P			
No. of stages	15	16	17	18	15	16	17	18
No. of states	1205	3598	13765	19861	3348	19146	22382	27742
CPU times (sec)	0.29	0.85	3.71	6.00	1.52	12.8	13.9	19.0
Memory usage (MB)	1.24	3.27	12.0	18.8	1.82	10.9	12.8	16.9

can_fire_last(s, t, t_p, T_D) checks all enabled transitions t'', and returns true if it is possible that t_p can fire later than any other descendant transitions of t'' that make t enabled. can_fire_last(s, t, t_p, T_D) is similar to necessary, but can_fire_last(s, t, t_p, T_D) considers all t'', while necessary checks some selected paths using minset.

5 Experimental Results

We have naively implemented the proposed method in the C language. Here, we demonstrate the verification of the STARI example [15, 16, 14] by using LTN models. In these experiments, the time Petri net models used in [14] are just replaced with LTN models, e.g., a NOR gate is modeled as shown in Figure 2(b) instead of Figure 2(a). The remaining verification settings are not changed.

In Table 1, the column labeled "Partial" shows the number of generated states, CPU times (Pentium III, 866MHz, 360MB, on VMware), and memory amount required for the verification of various sizes of STARI circuits. For comparison, the results by the total order algorithm where the set of all firable transitions is used as a ready set are also shown in the column labeled "Total". The results show a significant performance improvement of the partial order reduction algorithm over the total order algorithm.

Table 2 shows the performance comparison between the level-oriented method and the transition-oriented method. For this experiment, VINAS-P[17], which works for time Petri net models, is used as the transition-oriented method. Both methods use a partial order reduction algorithm. Since the LTN models are much simpler than the time Petri net models as shown in Figure 2, our naive implementation outperforms VINAS-P.

6 Conclusion

This paper proposes a level-oriented model, LTN, for formal verification that naturally models the behavior of asynchronous circuits. This new model allows for the specification of causality through both transitions and signal values. This paper also develops a partial order verification algorithm for this new model. In particular, the ready set construction is enhanced to be aware that disablings can now occur not only as a result of conflict in the net but also through the change of signal values in the level. The calculation of true parents in disjunctive conditions must also be considered in the calculation of the ready set. Finally, the necessary set construction used in the ready set calculation must be updated to allow for the recursion to proceed from a condition to the transition that assigns to the variables used in the condition. The zone construction used by the timing analysis algorithm must also be enhanced. In particular, true parents may now be found in conditions and more care must be taken in deciding when transitions can be safely pruned from the zone. This updated algorithm has been implemented and applied to the the timed circuit benchmark, STARI, and it has been found to outperform a verifier based on the time Petri net model.

References

[1] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT press, 1988.

[2] J. Ebergen and R. Berks. VERDECT: A verifier for Asynchronous Circuits. *IEEE TCCA Newsletter*, 1995.

[3] Oriol Roig, Jordi Cortadella, and Enric Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. *LNCS 935 Application and Theory of Petri Nets 1995*, pages 374–391, 1995.

[4] K. L. McMillan. Trace theoretic verification of asynchronous circuits using unfoldings. *LNCS 939 Computer aided verification*, pages 180–195, 1995.

[5] T. Yoneda and T. Yoshikawa. Using partial orders for trace theoretic verification of asynchronous circuits. *Proc. of Second International Symposium on*

Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)
0-7695-1852-4/02 $17.00 © 2002 IEEE

Advanced Research in Asynchronous Circuits and Systems, pages 152–163, 1996.

[6] R. Alur and D. Dill. The Theory of Timed Automata. *LNCS 443 (17th ICALP)*, pages 322–335, 1990.

[7] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. *Proc. of CONCUR98*, pages 485–500, 1998.

[8] Marius Minea. *Partial order reduction for verification of timed systems*. PhD thesis, Carnegie Mellon University, 1999.

[9] W. Belluomini, C. J. Myers, and H. P. Hofstee. Timed Circuit Verification Using TEL Structures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 20(1):129–146, January 2001.

[10] Eric G Mercer, Chris J. Myers, Tomohiro Yoneda, and Hao Zheng. Modular Synthesis of Timed Circuits using Partial Orders on LPNs. *Proc. of TPTS2002*, 2002.

[11] Y. Oguro, O. Okano, and T. Yoneda. Verification of asynchronous circuits including data-paths. *IEICE Technical Report (in Japanese) FTS2000(9)*, pages 65–72, 2000.

[12] Eric G Mercer. *Correctness and Reduction in Timed Circuit Analysis*. PhD thesis, University of Utah, 2002.

[13] B. Zhou, T. Yoneda, and C. Myers. Framework of Timed Trace Theoretic Verification Revisited. *Proc. of 10th Asian Test Symposium*, pages 437–442, 2001.

[14] Tomohiro Yoneda and Hiroshi Ryu. Timed trace theoretic verification using partial order reduction. *Proc. of Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 108–121, 1999.

[15] S. Tasiran and R. Brayton. STARI: A case study in compositional and hierarchical timing verification. *LNCS 1254 Computer Aided Verification*, pages 191–201, 1997.

[16] W. Belluomini and C. Myers. Verification of timed systems using POSETs. *LNCS 1427 Computer Aided Verification*, pages 403–415, 1998.

[17] http://yoneda-www.cs.titech.ac.jp/~yoneda/pub.html.

Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)
0-7695-1852-4/02 $17.00 © 2002 IEEE