Automatic Derivation of Timing Constraints by Failure Analysis

Tomohiro Yoneda*[1], Tomoya Kitai[2], and Chris Myers**[3]

[1] National Institute of Informatics
Tokyo 101-8430, Japan
yoneda@nii.ac.jp
[2] Tokyo Institute of Technology
Tokyo 152-8552, Japan
kitai@yt.cs.titech.ac.jp
[3] University of Utah
Salt Lake City UT 84112, USA
myers@ece.utah.edu

Abstract. This work proposes a technique to automatically obtain timing constraints for a given timed circuit to operate correctly. A designated set of delay parameters of a circuit are first set to sufficiently large bounds, and verification runs followed by failure analysis are repeated. Each verification run performs timed state space enumeration under the given delay bounds, and produces a failure trace if it exists. The failure trace is analyzed, and sufficient timing constraints to prevent the failure is obtained. Then, the delay bounds are tightened according to the timing constraints by using an ILP (Integer Linear Programming) solver. This process terminates when either some delay bounds under which no failure is detected are found or no new delay bounds to prevent the failures can be obtained. The experimental results using a naive implementation show that the proposed method can efficiently handle asynchronous benchmark circuits and nontrivial GasP circuits.

Keywords: Trace theoretic verification, Failure analysis, Timed circuits, Timing constraints.

1 Introduction

In order to obtain high performance systems, it is necessary to design circuits with aggressive and complex sets of timing constraints. GasP circuits [1] are a prime example of such highly *timed circuits*, i.e., circuits that don't work as expected, unless strict timing constraints on delay parameters are satisfied. In particular, the correctness of GasP circuits depends on the fact that (1) no hazards occur, (2) hold time constraints are satisfied for some signal transitions, and (3) short circuits caused by turning on all transistors in the path between the power supply and ground either never occur or occur only for a very short time. It is, however, not easy to check if the circuit satisfies all these constraints

* This research is supported by JSPS Joint Research Projects.
** This research is supported by NSF CAREER award MIP-9625014, NSF Japan Program award INT-0087281, and SRC grant 99-TJ-694.

D. Brinksma and K. G. Larsen (Eds.): CAV 2002, LNCS 2404, pp. 195–208, 2002.

by simulation or static timing analysis due to the complexity of the timing constraints. Therefore, formal verification is essential.

This work uses a formal verification tool VINAS-P [2]. VINAS-P is based on a timed version of trace theoretic verification [3], and time Petri nets are used for modeling both specifications and circuits. VINAS-P checks safety properties. Bad behavior such as a hazard, hold time violation, and a short circuit can be detected as safety failures. VINAS-P uses partial order reduction which explores only a reduced state space that is sufficient to detect failures, which enables us to verify much larger circuits than a traditional total order method.

Although a formal verifier is very effective to prove a given circuit is correct with respect to the specification, for an incorrect circuit, it simply generates a failure trace. In the case of VINAS-P, it shows for a failure trace a waveform of selected signals. This is useful to understand what is going on in a circuit, but, it is not easy to see why the failure occurs, or how the failure can be eliminated. When we tried to verify the GasP circuits, failure traces were actually produced again and again. Although almost all these failures are caused by incorrect delay settings, obtaining the appropriate delays or conditions for them is a difficult problem. This motivates this work, which proposes a way to obtain sufficient timing conditions on delays for correct behavior of timed circuits by analyzing failure traces produced by the verifier.

In the proposed method, several delay parameters are selected to be examined, and initially, some large integer bounds are set to them. Then, the model is verified. If a failure trace is provided by the verifier, then our algorithm analyzes it, and suggests a set of candidates for additional timing constraints. Those timing constraints are sorted using heuristics, and the most appropriate one is chosen by the algorithm. The rest of the constraints are used when backtracking occurs. The selected timing constraint is added to the initial timing constraints, meaning the delay bounds are tightened. Then, an ILP (Integer Linear Programming) solver is invoked to update the delay bounds. This new set of delay bounds are used for the next verification run. This process of verification, analysis of failure traces to obtain timing constraints, and updating the delay bounds are fully automatic, and it is repeated until verification succeeds or no consistent timing constraints are found. Integer delay bounds and ILP are used in order to guarantee the termination of this process.

The rest of this paper is organized as follows. Section 2 refers to related works. Section 3 briefly introduces the verification method. Section 4 shows an example to explain the proposed method intuitively. In Section 5, the algorithms to analyze a failure trace and obtain timing constraints to eliminate the failure are proposed. The heuristics used for performance improvement are also shown there. Section 6 shows experimental results using a naive implementation. Finally, Section 7 summarizes the discussion.

2 Related Works

The same problem discussed in this paper is solved by two different but similar approaches. In [4], Negulescu proposes a method where a timed circuit is represented by an untimed model, called a process, and untimed state space enumeration is done. When a failure is detected, they analyze it by hand and

construct a new model that avoids the failure. This process of untimed verification and reconstruction of the model is repeated until no failure is detected or model reconstruction fails due to inconsistency. Another approach is proposed in [5,6]. This approach also uses untimed models and untimed verification. In this approach, all possible failures of a circuit are generated by one state enumeration, and then timing constraints are obtained automatically by analyzing the state graph. The constraints obtained by this approach are not those on delays but those on ordering of signal transitions. Thus, their goal is slightly different from ours.

Another work that we need to mention is a verification of timed systems using relative timing method, which is proposed in [7]. Its goal is to verify timed circuits, not to obtain timing constraints. But, in their method, a detected failure is checked if it is legal with respect to the given delay bounds, and if so, a new model that excludes the failure is reconstructed. Verification and reconstruction are repeated similarly to Negulescu's method, but automatically. While it may be possible to combine this work and Negulescu's work to achieve the same result as ours, it is not clear how effective this would be since it has not been attempted.

The biggest difference between these works and our work is that only our method uses timed state space enumeration. The authors of the above works claim that the advantage of their works is that the verification of timed systems can be reduced to that of untimed systems. It is apparent that the complexity of untimed verification is much smaller than that of timed verification. Our claim is, however, that a huge number of failures may be detected if a timed circuit is analyzed as an untimed circuit, i.e., many but unrealistic failure traces can be produced by the untimed analysis. This makes the cost to obtain timing constraints fairly large. If the initial delay bounds can be suitably reduced to realistic ones, our method may work more efficiently. Probably, the only way to compare both approaches is to implement our idea and to compare the results for many examples. This is one of the goals of this paper.

Another difference is in adding timing constraints. Our method uses updated delay bounds. Thus, the cost of each verification run is almost the same. On the other hand, in the method proposed in [4] and [7], an additional timing constraint is represented by a process or a transition system, and the composition of the original model and the model for the additional timing constraint is verified in the next run. It is possible that this more complicated model may require more BDD nodes and increase the verification cost. The method in [5] does not suffer from this problem, because no model reconstruction is done. However, their method does not obtain constraints on delays but ordering of signal transitions, and hence, it seems difficult to verify, for example, hold time violation. In order to obtain constraints on delays, model reconstruction or a re-verification step (in our case) is necessary in each iteration, because there are potentially many constraints that eliminate a particular failure, and searching appropriate combinations of constraints to eliminate all possible failures step by step with backtracking is much easier than obtaining all possible combinations on the first try. For this reason, our problem cannot be modeled by a uniform ILP problem.

It's also necessary to mention that there are many works[8,9,10, and others] to verify timed systems using timed automata. Although this work uses time Petri nets to model timed circuits, because a tool based on them is available for us, we

believe that the technique proposed in this paper can be easily applied to timed automaton based tools. Furthermore, although our tool uses the DBM analysis to handle real-time constraints, the proposed technique can also be applied to discrete-time analysis methods.

3 Verification Method

The underlying verification method used in this work is the timed extension of trace theoretic verification [3]. In our method, each circuit element, called a module, is modeled by a time Petri net.

A 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*. In this work, it is assumed that these times are integers. 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.

A module is defined as (I, O, N), where I and O are sets of input and output wires, respectively, and N is a time Petri net. A firing of a transition changes the value of a wire that is related to the transition, and the direction of change ($0 \rightarrow 1$ or $1 \rightarrow 0$) is represented by $+$ or $-$ in its name. A transition that is related to an output wire of the module is called an *output transition*. An *input transition* is defined similarly.

A timed circuit is modeled by a set of modules. In a set of modules, an input transition fires only in synchronization with the corresponding output transition in some different module. Thus, the earliest and latest firing times of an input transition is considered to be $[0, \infty]$. If an output transition is firable and every corresponding input transition is disabled in a module, the state is called a *failure state*, and the verifier reports a *failure trace*, which is a sequence of all transitions fired between the initial state and the failure state.

A specification is also modeled as a module. If a circuit behaves differently from its specification, an output from a circuit module cannot be accepted by the specification, and it is detected as a failure. In addition, bad behavior such as a hazard, hold time violation, and a short circuit can be detected as failures inside circuit modules.

4 A Small Example

Let's consider a circuit shown in Figure 1(a), where the delay bounds of the inverter and OR gate are $[d_{inv}, D_{inv}]$ and $[d_{or}, D_{or}]$. The initial state of this circuit is $(a, b, c, d) = (1, 0, 0, 0)$, and its behavior is expected as follows (See Figure 1(b)): When c is raised, d goes up. Then, a and c are lowered in this order. During these input changes, the circuit keeps d high. Finally, when a is raised again, d goes down, and the circuit goes back to the initial state. Hence,

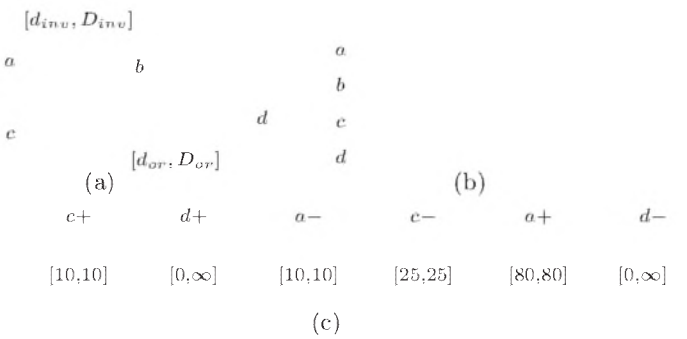

Fig. 1. A circuit and its environment

Fig. 2. Two possible cases to prevent the failure

the environment of this circuit can be expressed by $(\{d\}, \{a, c\}, N_s)$, where N_s is a time Petri net shown in Figure 1(c) [1]. The delay bounds for $c+$ and $a-$ are [10,10], while those for $c-$ and $a+$ are [25,25] and [80,80], respectively. Note that d is an input of this environment and it fires in synchronization with the circuit output.

Assume that the following initial constraints for the circuit delay bounds are given.

$$\begin{array}{c} 5 \le d_{inv} \le 50, \quad 5 \le D_{inv} \le 50, \quad 5 \le d_{or} \le 50, \quad 5 \le D_{or} \le 50, \\ d_{inv} + 2 \le D_{inv} \le d_{inv} + 30, \qquad d_{or} + 2 \le D_{or} \le d_{or} + 30 \end{array} \tag{1}$$

The constraints of the form $d_{inv}+2 \le D_{inv}$ are used to avoid tight delay bounds, and those of the form $D_{inv} \le d_{inv} + 30$ are for reducing the state space. These and the lower bounds are also important to avoid imbalanced delay assignment such as assigning total delay to one gate and zero to the others. Actually, these initial delay bounds should be determined depending on the device technology used to implement the circuits. Now, the problem to be solved is to find some delay bounds, satisfying the above constraints, under which the circuit behaves correctly with its environment. Although it is desired that maximal possible delay bounds are found, it is beyond the scope of this paper.

The first step of our algorithm is to obtain initial delay bounds from (1) using a ILP solver. In this case, they are

$$d_{inv} = 5, \; D_{inv} = 35, \; d_{or} = 5, \; D_{or} = 35.$$

[1] More precisely, this is defined as a *mirror* of a specification, where their input set (output set) is equal to the output set (input set) of the circuit.

The details about the ILP solver and the objective function used are mentioned in Section 6. Using these delay bounds, the first verification run is done, and the following failure is detected: [2]

$$c+;\ d+;\ a-;\ c-;\ b+$$

This failure means that after $c-$, the OR gate tries to lower its output d because b is low at that time. But, before its output change, $b+$ occurs. This violates a property called semi-modularity, and is considered to produce a hazard. This failure can be prevented, if (a) $b+$ occurs later than the output change $d-$, or (b) $b+$ occurs before the input change $c-$ (See Figure 2(a) and (b)). Note that the failure is prevented in case (b), because the output of the OR gate is stable during these input changes. Suppose that our algorithm first tries case (a). In order to obtain the constraint for (a), the algorithm examines the casuals of $b+$ and $d-$. $b+$ is caused by $a-$, while $d-$ is caused by $c-$ and $c-$ is caused by $a-$ in the environment. Hence, to make $b+$ occur later than $d-$, the following constraint is necessary.

$$25 + D_{or} < d_{inv} \tag{2}$$

Note that the largest delay is used for the OR gate, while the smallest delay is used for the inverter. This ensures the above ordering ($d-;\ b+$) even in the worst case.

For constraints (1) and (2), the ILP solver gives the delay bounds

$$d_{inv} = 33,\ D_{inv} = 50,\ d_{or} = 5,\ D_{or} = 7,$$

and the second verification run with these delay bounds produces the following failure.

$$c+;\ d+;\ a-;\ c-;\ d-$$

This failure occurs, because the circuit produces $d-$ although it is not expected in the environment (i.e., $d-$ is not enabled after $c-$). In other words, to prevent this failure, $d-$ should be prevented. This is possible if $b+$ occurs before $c-$. Again, the algorithm checks their casuals, and finds that both $b+$ and $c-$ are caused by $a-$. Hence, the following constraint is obtained.

$$D_{inv} < 25 \tag{3}$$

For constraints (1), (2), and (3), however, the ILP solver gives no solution due to inconsistency.

Now, the algorithm backtracks to the most recent selection point, and chooses case (b) instead. This constraint is actually the same as the above one, and constraint (3) is obtained. For constraints (1) and (3), the ILP solver gives

$$d_{inv} = 5,\ D_{inv} = 24,\ d_{or} = 5,\ D_{or} = 35,$$

and the third verification run reports no failure. Hence, the above delay bounds are the solution of our problem.

[2] Every gate is modeled by a time Petri net [3], and it contains internal transitions other than input or output transitions. A failure trace includes internal transitions, but here, they are omitted for simplicity.

The main technical issue of our algorithm is to automatically obtain a constraint to prevent the given failure by analyzing the failure trace and the structure of the Petri nets. Another issue is that the correctness of the algorithm depends on the backtracking. In the above example, one backtrack occurs. Many backtrackings, of course, decreases the performance of the algorithm. Our algorithm uses a heuristic to choose appropriate constraints, which is simple, but very effective. These issues are discussed in the following section.

5 Failure Analysis

This section presents the algorithm that is used to perform analysis to derive sufficient timing constraints to avoid failures.

5.1 Finding A and B Events

When a failure trace is given, our algorithm first finds two events, called event A and event B, such that the failure is caused because event A occurs before event B, and that the failure may be prevented by firing event B before event A. For a failure trace, there can exist several event A's and B's. In the above example, for case (a), A is $b+$ and B is $d-$, and for case (b), A is $c-$ and B is $b+$. In order to handle cases where event B may not be even enabled, event A and B are extended so that they have an offset. That is, an *AB-candidate* with respect to a failure trace $\mathcal{F}$ is a three tuple $\langle t_A, t_B, \mathit{off} \rangle_{\mathcal{F}}$, where t_A is a transition that fires in $\mathcal{F}$, t_B is a transition that is enabled in the state where t_A fires, such that firing t_B certainly *off* time units earlier than t_A may be able to prevent $\mathcal{F}$. $\mathcal{F}$ is omitted from this notation if there is no confusion.

Let's consider modules M_1, M_2 shown in Figure 3 and their failure trace $\mathcal{F} = a+;\ t_4;\ t_2;\ t_1;\ b+(out)$. This failure trace starts when an output transition $a+(out)$ of M_2 fires in its initial marking $\mu_0 = \{p_0, p_5\}$ as well as the corresponding input transition $a+(in)$ of M_1. The failure occurs in a marking $\mu_3 = \{p_3, p_4, p_7\}$ because $b+(out)$ of M_1 fires before its input transition $b+(in)$ of M_2 becomes enabled. Thus, one way to prevent this failure is to fire t_6 before $b+(out)$. Note that an input transition is assumed to have $[0, \infty]$ bound, and so it becomes ready to fire immediately when it is enabled. In this example, however, t_6 is not yet enabled when $b+(out)$ of M_1 fires. Thus, the net is traversed upward, and an enabled transition t_5 is found. Since t_6 takes D_6 time units to fire in a worst case, it is necessary to fire t_5 D_6 time units earlier than $b+(out)$. Hence, $\langle b+(out), t_5, D_6 \rangle$ is obtained. This AB-candidate is computed by $\mathsf{force\_fire}(b+(in), b+(out), 0, \mu_3, \emptyset)$, where $\mathsf{force\_fire}(t, t_A, \mathit{off}, \mu, T_D)$ obtains a set of AB-candidates in a marking μ to force t to fire certainly *off* time units earlier than t_A without firing transitions in T_D, and it is defined as follows.

1. If $t \in T_D$, then $\mathsf{force\_fire}(t, t_A, \mathit{off}, \mu, T_D) = \emptyset$. T_D is used to terminate looping.
2. Otherwise, if t is enabled in μ, then

$$\mathsf{force\_fire}(t, t_A, \mathit{off}, \mu, T_D) = \{\langle t_A, t, \mathit{off} \rangle\}.$$

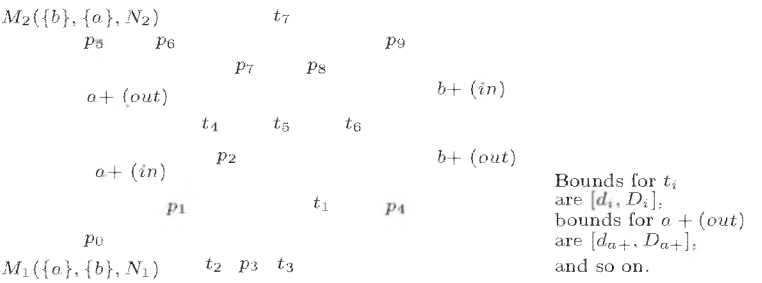

Fig. 3. An example of a module set

3. Otherwise, for some empty place $p \in \bullet t - \mu$,

$$\mathsf{force\_fire}(t, t_A, \mathit{off}, \mu, T_D) = \bigcup_{t' \in \bullet p} \mathsf{force\_fire}(\mathsf{out\_trans}(t'), t_A, \mathit{off} + \mathsf{Lft}(t), \mu, T_D \cup \{t\}),$$

where $\mathsf{out\_trans}(t')$ is the output transition that corresponds to t' (if t' is an output transition, then $\mathsf{out\_trans}(t') = t'$), and $\mathsf{Lft}(t)$ is the latest firing time of t. Note that it is sufficient to check some empty source place p of t because at least p needs a token in order to enable t. On the other hand, all source transitions of p should be checked, because it is unknown which source transition produces a token to p.

There are, however, other ways to prevent the above failure. For example, if t_3 fires before t_1, this failure is prevented, because the output transition $b+(\mathit{out})$ is no longer enabled. Furthermore, if t_7 fires before t_4 and $b+(\mathit{out})$, this failure is prevented. The method used in our work to cover all these cases is to try every transition t_c that lost the chance to fire in the failure trace, i.e., our method obtains every AB-candidate for firing transition t_c such that $t_c \in \mathsf{conflict}(t)$ where t is a transition that fired in the failure trace and $\mathsf{conflict}(t)$ is a set of transitions that are in conflict with t. Since this method may produce unnecessary AB-candidates, removing them is probably necessary in order to improve the performance, but this is left as future work. Hence, the following $\mathsf{obtain\_AB}(\mathcal{F})$ obtains a set of all AB-candidates for a failure trace $\mathcal{F}$, where $\mathsf{in\_trans}(t, M)$ is a set of input transitions of module M that correspond to output transition t, M_{in} is the module whose input transition causes a failure, $l = |\mathcal{F}|$, t_i is the i-th transition in $\mathcal{F}$ (i.e., t_{l-1} is the failure transition), and μ_i is the marking where t_i fires (i.e., μ_0 is the initial marking).

$$\mathsf{obtain\_AB}(\mathcal{F}) = \bigcup_{t' \in \mathsf{in\_trans}(t_{l-1}, M_{in})} \mathsf{force\_fire}(t', t_{l-1}, 0, \mu_{l-1}, \emptyset) \cup \bigcup_{i=0}^{l-2} \Big(\bigcup_{t' \in \mathsf{conflict}(t_i)} \mathsf{force\_fire}(\mathsf{out\_trans}(t'), t_i, 0, \mu_i, \emptyset) \Big)$$

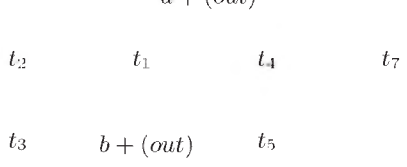

Fig. 4. Timing relations implied by the failure trace $\mathcal{F}$

5.2 Obtaining Constraints

Once AB-candidates are found, the next step is to construct timing constraints for each AB-candidate. This is done based on the timing relations implied by the given failure trace. A failure trace gives two kinds of timing relations, called *causal relation* and *preceding relation.*

If transition u is the unique parent of transition t, i.e., the firing of u causes t to become enabled, the firing time of t, denoted by $T(t)$, must satisfy the following relation.

$$\mathsf{Eft}(t) \leq T(t) - T(u) \leq \mathsf{Lft}(t)$$

This is a causal relation. If t has two or more parents $u_1, u_2, \cdots$, the verification algorithm chooses one parent, say u_p, that decides the firing time of t. Such a parent is called a *true parent.* Since a true parent must fire later than the other parents in order to actually cause its child transition, the following relation is also necessary besides the above causal relation.

$$T(u_1) \leq T(u_p), \quad T(u_2) \leq T(u_p), \quad \cdots$$

These are called preceding relations. Furthermore, if two or more transitions $t_1, t_2, \cdots$ are in conflict, and t_k wins the conflict, then the following relation is necessary to express that t_k fires earlier than any other conflicting transitions.

$$T(t_k) \leq T(t_1), \quad T(t_k) \leq T(t_2), \quad \cdots$$

These are also preceding relations. Precisely, this relation is necessary for all transitions in a ready set [3], which is a set of transitions that should be interleaved in the state.

Consider again the modules shown in Figure 3 and the failure trace $\mathcal{F} = a+;\ t_4;\ t_2;\ t_1;\ b+(out)$. The timing relations implied by this failure trace can be illustrated as shown in Figure 4. In this figure, which is called a *failure graph*, a node represents a transition that fires or gets enabled in the failure trace. A normal arrow from u to t indicates the causal relation (i.e., $\mathsf{Eft}(t) \leq T(t) - T(u) \leq \mathsf{Lft}(t)$), while a dotted arrow indicates the preceding relation (i.e., $T(u) \leq T(t)$).

Now, consider an AB-candidate $\langle b+(out), t_5, D_6 \rangle$ to construct its timing constraints for firing t_6 certainly earlier than $b+(out)$. The first step to obtain the constraints is to find the common ancestor of t_5 and $b+(out)$ in the failure graph. In this example, it's $a+(out)$. This means that $a+(out)$ determines the firing times of both t_5 and $b+(out)$, and so the constraints should be related to

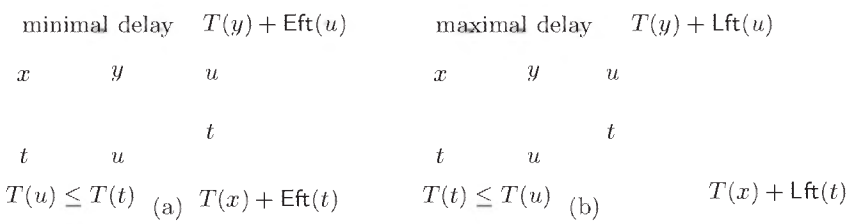

Fig. 5. Paths by preceding relation

the delays between $a + (out)$ and those two events. Next, in order to guarantee the above relation between t_6 and $b + (out)$, the maximal delay from $a + (out)$ to t_5 plus D_6 must be smaller than the minimal delay from $a + (out)$ to $b + (out)$. From the causal relation of $\mathcal{F}$, this is expressed as follows.

$$D_4 + D_5 + D_6 < d_1 + d_{b+}$$

Note that bounds for t_i are denoted by $[d_i, D_i]$, bounds for $a + (out)$ are $[d_{a+}, D_{a+}]$, and so on. In addition to the above constraint, the effect of the preceding relation should be considered. When computing minimal delay up to t, suppose that there is a preceding relation $T(u) \leq T(t)$ as shown in Figure 5(a). Due to this constraint, if u fires late enough, the earliest firing time of t is not decided by $T(x) + \mathsf{Eft}(t)$, but decided by $T(y) + \mathsf{Eft}(u)$. This means that the path shown by the dotted arrow in the figure should also be considered for the minimal delay path. Since it is difficult to check if u certainly fires late enough, both paths (i.e., $x \to t$ and $y \to u \to t$) need to be considered. Similarly, for the maximal delay computation, the dotted arrow in Figure 5(b) should be considered. Hence, another constraint like

$$D_7 + D_5 + D_6 < d_1 + d_{b+}$$

is also necessary.

5.3 Heuristics to Select Constraints

Since the algorithms shown in the previous sections obtain constraints for considering all possibilities to prevent the given failure, many constraints are often generated, Thus, it is very important to select an appropriate one from them. This subsection shows simple heuristics for this purpose.

Let $v(d)$ be a value assigned to a delay d by the ILP solver for the most recent verification run, and for an expression $E = d_1 + d_2 + \cdots$, let $v(E)$ be $v(d_1) + v(d_2) + \cdots$. A *weight* of a constraint $L < R$ is $v(L) - v(R)$, where L and R are expressions. The idea is that the weight of a constraint implies how much effort is necessary to satisfy the constraint based on the current delay assignment. For example, for the current delay assignment such as $v(d_1) = 10$, $v(d_2) = 50$, and $v(D_3) = 60$, it may be easier to satisfy a constraint $D_3 < d_2$ rather than to satisfy $D_3 < d_1$, because d_2 should be increased by more than 10 for the former, while d_3 should be increased by more than 50 for the latter. This

is represented by the weights 10 and 50, respectively. Note that a constraint with negative weight is illegal, because such a constraint is supposed to be already satisfied under the current delay assignment, and it cannot prevent the given failure.

If a constraint that is too strong is selected, an inconsistency may be detected after several verification runs, and backtracking occurs. On the other hand, even if a constraint that is too weak is selected, a stronger constraint can be added later to obtain a suitable constraint set. Hence, our heuristics select a constraint with the smallest nonnegative weight. For the example shown in Section 4, the weight of constraint (2) is 55, and that of (3) is 10. Hence, if this heuristic is used, case (b) is selected first, and no backtracking occurs.

5.4 Overall Procedure

The whole procedure that repeats the verification runs and adds new constraints is shown in Figure 6. This procedure takes two inputs, M and *con_set*. M is a set of time Petri nets representing the circuit and its specification. When the procedure is called for the first time, the initial constraints for the circuit delay bounds like (1) in Section 4 is set to *con_set*. This procedure first calls an ILP solver (line 3). Currently, we use a public domain ILP solver called lp_solve (ver 3.1a, ftp://ftp.ics.ele.tue.nl/pub/lp_solve/). An ILP solver computes an optimal integer assignment to variables for maximizing or minimizing an objective function under a given set of constraints. For delays $d_1, D_1, d_2, D_2, \cdots$ where d_i is a lower bound of the delay and D_i is an upper bound, our algorithm uses the following objective function f and tries to maximize it.

$$f = (D_1 - 2d_1) + (D_2 - 2d_2) + \cdots$$

From our experience, the most suitable solutions such that the difference between lower bounds and upper bounds are large and that lower bounds are fairly small are obtained by this objective function. *stat* in line 3 indicates "infeasible", if the constraint set is inconsistent. In this case, the procedure returns with "impossible" for backtracking (line 4). Otherwise, *bounds* contains an optimal assignment to the delay bounds. In line 5, the bounds of M are modified according to this delay assignment, and M' is obtained. This M' is used for the verification in line 6. If the verifier returns "success", this means that a set of timing constraints under which the circuit works as expected are obtained, and so, the procedure terminates (line 7). Otherwise, the verifier produces a failure trace *failure*. In line 8, this *failure* is analyzed as mentioned in the previous subsections, and a set of new timing constraints are obtained. Those timing constraints are sorted based on their weights (line 9), and each constraint *con* with a nonnegative weight is added to *con_set* in this order for the recursive call of "obtain_timing_constraints" (line 11). If it returns, it means that no solution is obtained under *con_set* $\cup$ {*con*}, and so, the next constraint in the *new_con'* is tried by the foreach loop (line 10 and line 11). If every constraint causes inconsistency, the procedure returns with "impossible" for backtracking (line 12).

By selecting a constraint with a nonnegative weight, it is guaranteed that the constraint certainly reduces the space of the delay bounds. Therefore, since the earliest and latest firing times are integer, this procedure always terminates.

```
1 : obtain_timing_constraints(M, con_set)
2 : begin
3 :   (stat, bounds) = ILP(con_set);
4 :   if (stat == infeasible) then return(impossible);
5 :   M' = modify_bounds(M, bounds);
6 :   (stat, failure) = verify(M');
7 :   if (stat == success) then exit(success);
8 :   new_con = analyze_failure(failure);
9 :   new_con' = sort(new_con);
10 :  foreach con ∈ new_con'
11 :    if weight(con) ≥ 0 then obtain_timing_constraints(M, con_set ∪ {con});
12 :  return(impossible);
13 : end
```

Fig. 6. Overall procedure

On the other hand, when the procedure terminates with "impossible", is it really impossible to eliminate the failure? If so, the procedure is called *complete*. In order to prove its completeness, it is necessary to show that the algorithm to find an AB-candidate covers all cases to eliminate failures, and that the constraints obtained are not unnecessarily strict. This is not yet proven formally. The selection of objective function as well as errors in the ILP solutions certainly affect the performance (i.e., the number of backtrackings) and the quality of the results (i.e., the width of the delay bounds), but we do not believe that the completeness is affected by them.

6 Experimental Results

In order to demonstrate the proposed method, the VINAS-P verifier has been modified so that it produces a set of timing constraints for a detected failure trace. This program corresponds to lines $6 \cdots 8$ in Figure 6. Then, a Perl script has been developed to naively implement the rest of the procedure.

In this section, two sets of experimental results are shown. The first set of experiments have been done using some asynchronous benchmark circuits from [6]. The second and third columns of Table 1 show the number of signals and the number of gates in each circuit. "#timed states" shows the number of timed states in the circuits with the final bounds (i.e., the circuits that pass verification). The next two columns show the number of verification runs and the number of backtracks needed to obtain the final constraint sets. The CPU times for the overall procedure are shown in the column "CPU"(all CPU times are shown in seconds). The column "CPU-[6]" is quoted from [6], where the experiments were done on a 450MHz 1GB Ultra SPARC60 machine. According to the authors of that paper, the data comes from a proof-of-concept prototype that is not yet optimized for run-time and thus does not incorporate many of the known speed-up techniques and optimizations for untimed analysis. In addition, the majority of the the run-time is taken up in the process for optimizing constraint sets that can be made much more efficient. Our experiments have been performed on a Pentium II 333MHz, 128MB Linux machine, and as mentioned,

Table 1. Experimental results (1)

name	#signals	#gates	#timed states	#verify	#backtracks	CPU	CPU-[6]
alloc-outbound	15	11	85	4	0	1.36	13.08
mp-forward-pkt	13	10	57	3	0	0.93	0.89
dff	8	6	67	6	0	1.48	27.17
sbuf-send-pkt2	17	13	113	7	0	2.69	69.97
converta	14	12	98	7	0	2.36	113.12
ram-read-sbuf	22	16	161	7	0	3.08	127.98

Table 2. Experimental results (2)

name	#signals	#gates	#timed states	#verify	#backtracks	CPU	CPU (last)
gasp4	27	32	817	9	0	2.07	0.11
gasp8	51	64	65147	10	0	752.17	717.77
square9	82	81	3017	11	2	21.26	2.04

our current implementation is also very naive. Thus, we consider that these data demonstrate that the performance of our method based on time analysis is at least comparable to those of their method based on untimed analysis.

If the sorting by the constraints' weights (line 9 of Figure 6) is turned off, 10 verification runs and 5 backtrackings are needed for the "alloc-outbound" circuit. This shows the effectiveness of the heuristics shown in Section 5.3.

The second set of experiments [3] use several GasP circuits shown in [1] and [11]. These circuits have fairly large state spaces, but our method can handle them as shown in Table 2. Almost all CPU times are spent for the final verification of the correct circuits as shown in the last column (CPU (last)), and the process to obtain the timing constraint sets is performed within a rather short time. These experiments have been performed on a Pentium III 1GHz, 2MB Linux machine. In these experiments, the CPU times for ILP is negligible compared with those for state space enumeration. Thus, from a performance point of view, using ILP instead of LP is not too costly.

7 Conclusion

This paper describes a new method for the derivation of timing constraints that guarantee the correctness of timed circuit implementations. This approach uses an automatic technique in which a failure trace is analyzed to find pairs of events and obtain associated new timing constraints that can eliminate the failure trace. This method has been automated around the VINAS-P tool, and our initial verification results are very promising.

In the future, we plan to develop better heuristics to avoid generating useless AB-candidates. We also plan to perform a formal analysis to show that our method is complete in that when no constraints can be found, no solution can exist.

[3] The source files and results of these experiments can be downloaded from http://yoneda-www.cs.titech.ac.jp/~yoneda/tcs-data/data.tar.gz.

Acknowledgement

The authors would like to thank Peter Beerel and Hoshik Kim for helping us to understand their method and giving their latest experimental results, and to thank Bill Coates and Ian Jones for helpful comments to model GasP circuits.

References

1. Ivan Sutherland and Scott Fairbanks. GasP: A minimal FIFO control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53. IEEE Computer Society Press, March 2001. 195, 207
2. http://yoneda-www.cs.titech.ac.jp/~yoneda/pub.html. 196
3. Tomohiro Yoneda and Hiroshi Ryu. Timed trace theoretic verification using partial order reduction. In *Proc. of Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 108–121, 1999. 196, 198, 200, 203
4. Radu Negulescu and Ad Peeters. Verification of speed-dependences in single-rail handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 159–170, 1998. 196, 197
5. Hoshik Kim. Relative timing based verification of timed circuits and systems. In *Proc. International Workshop on Logic Synthesis*, June 1999. 197
6. Hoshik Kim, Peter A. Beerel, and Ken Stevens. Relative timing based verification of timed circuits and systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 115–124, 2002. 197, 206, 207
7. Marco A. Peña, Jordi Cortadella, Alex Kondratyev, and Enric Pastor. Formal verification of safety properties in timed circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11. IEEE Computer Society Press, April 2000. 197
8. Rajeev Alur and David Dill. Automata for modeling real-time systems. *LNCS 600 Real-time: Theory in Practice*, pages 45–73, 1992. 197
9. Marius Bozga, Oded Maler, and Stavros Tripakis. Efficient Verification of Timed Automata Using Dense and Discrete Time Semantics. In *Proc. of 10th IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, LNCS 1703, pages 125–141, 1999. 197
10. Marius Minea. *Partial order reduction for verification of timed systems.* PhD thesis, Carnegie Mellon University, 1999. 197
11. Jo Ebergen. Squaring the FIFO in GasP. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 194–205. IEEE Computer Society Press, March 2001. 207