EFFICIENT VERIFICATION OF HAZARD-FREEDOM IN GATE-LEVEL TIMED ASYNCHRONOUS CIRCUITS

Curtis A. Nelson and Chris J. Myers

University of Utah
Salt Lake City, UT
{nelscu,myers}@vlsigroup.ece.utah.edu

Tomohiro Yoneda

National Institute of Informatics
Tokyo, Japan
yoneda@nii.ac.jp

ABSTRACT

This paper presents an efficient method for verifying hazard freedom in timed asynchronous circuits. Timed circuits are a class of asynchronous circuits that utilize explicit timing information for optimization throughout the entire design process. In asynchronous circuits, correct operation requires that there are no hazards in the circuit implementation. Therefore, when designing an asynchronous circuit, each internal node and output of the circuit must be verified for hazard-freedom to ensure correct operation. Current verification algorithms for timed asynchronous circuits require an explicit state exploration often resulting in state explosion for even modest sized examples. The goal of this work is to abstract the behavior of internal nodes and utilize this information to make a conservative determination of hazard-freedom for each node in the circuit. Experimental results indicate that this approach is substantially more efficient than existing timing verification tools. These results also indicate that this method scales well for large examples. It is capable of analyzing circuits in less than a second that could not be previously analyzed. While this method is conservative in that some false hazards may be reported, our results indicate that the number of false hazards is small.

1. INTRODUCTION

Timed circuits are a class of asynchronous circuits that use explicit timing information in circuit synthesis. This timing information, however rough the estimates may be, can potentially reduce the amount of circuitry that would be needed from a design that adheres to speed-independent constraints. The estimates for the timing can be verified once the design is mapped to a library and actual timing values are known. This simplification can lead to significant gains in circuit performance over asynchronous circuits designed without timing assumptions. This was demonstrated in the Intel RAPPID project in which an asynchronous instruction length decoder for an x86 processor was designed using timed circuits. It was found to be three times faster while using half the power of the comparable synchronous design [17].

While timed asynchronous circuits offer potential advantages over synchronous circuits such as faster operation and lower power, these advantages are often offset by the expense of the circuit overhead needed to eliminate *hazards*. Hazards are conditions generated by the structure of the circuit or timing relationships between inputs and propagation delays that can cause incorrect behavior.

This research is supported by SRC contract 2002-TJ-1024, an SRC Graduate Fellowship, NSF Japan Program award INT-0087281, and a JSPS Joint Research grant.

As synthesized hazard-free logic equations are mapped to a given gate library, new internal nodes are introduced in the circuit netlist. Each new internal node as well as the outputs of the circuit must be verified for hazard-freedom to ensure correct operation of the mapped circuit. This verification must be extremely efficient to allow for many alternative designs to be considered during technology mapping. Current timing verification algorithms [13, 3, 14, 10, 9, 18] often suffer from state explosion problems because each node in the circuit netlist is treated as a new state variable, potentially doubling the number of states.

There are numerous methods for verifying hazard-freedom in gate-level *speed-independent* circuits [5, 6, 2, 8, 7, 15, 16]. In speed-independent circuits, no timing assumptions are made about gates or the environment except that wire delays are negligible. An efficient verification method for *determinate* speed-independent circuits is proposed in [2]. *Determinate* speed-independent circuits allow input choice (conditionals) but not output choice (arbitration). The work in [2] reduces state explosion by examining individual behavior at each internal node and approximating this behavior for each state in the specification. The hazard-freedom of the circuit is then verified by examining this *cube approximation*. When the number of internal signals is high as compared with the number of primary inputs and outputs (a feature common of many circuit design styles), this cube approximation technique has the potential to substantially reduce the complexity of verification as demonstrated in the results shown in [2].

Abstraction of internal nodes to combat state explosion is also performed in [20, 19]. This work, however, is not directed at verification of hazard-freedom and requires the use of *Timed Petri Nets* for all design descriptions including the gates to be analyzed. While it is potentially possible that this work could be used to verify hazard-freedom, it is not known how successful it would be. It may be interesting in the future to explore combining this approach with the one proposed in this paper.

The goal of this paper is to extend the work in [2] to verify timed circuits. It is often the case that hazard conditions found in speed-independent circuits do not manifest as glitches in the real circuit implementation due to the actual timing behavior. The reason for this is that internal signals, once enabled, certainly do fire in some finite time. If the time evolution can be tracked in the state space, then it may be possible to identify the stability of internal signals. Using this *timed cube approximation*, a gate-level timed circuit can be rapidly analyzed for hazards. Experimental results show that this approach can be substantially faster than existing timing verifiers. Thus, the method presented in this paper has the potential to greatly increase the size of circuits that can be verified.

2. BACKGROUND TERMINOLOGY

The verifier described in this paper takes as inputs a *time Petri net* (TPN) defining the circuit and the behavior of the environment, and a *netlist* representing the circuit to be verified. The verification procedure also creates and uses a *state graph* to represent reachable timed states. This section describes each of these formally.

2.1. Time Petri nets

Our method uses TPN's [11] to model the possible input behaviors and the required output behaviors for timed circuits. Let W be a finite set of wires. The timed behavior of a circuit is modeled as sequences of rising and falling transitions on W. For any $w \in W$, $w+$ is a rising transition and $w-$ is a falling transition on the wire w. In the following definitions, let $\mathbb{Q}^+$ and $\mathbb{R}^+$ denote the sets of non-negative rational and non-negative real numbers, respectively. A W-labeled one-safe TPN is a directed bipartite graph described by the tuple $TPN = \langle W, T, P, F, M_0, s_0, l, u, L \rangle$ where:

- $W = I \cup O$ is the set of wires where I is the set of input wires and O is the set of output wires;
- T is the set of transitions;
- P is the set of places;
- $F \subseteq (T \times P) \cup (P \times T)$ is the flow relation;
- $M_0 \subseteq P$ is the initial marking;
- $s_0 \subseteq W$ is the set of wires that are initially high;
- $l : T \rightarrow \mathbb{Q}^+$ is the lower timing bound function;
- $u : T \rightarrow \mathbb{Q}^+ \cup \{\infty\}$ is the upper timing bound function;
- $L : T \rightarrow W$ is the labeling function.

The state of a TPN is a pair $\langle M, D \rangle$ where M is the current marking (i.e., the subset of places that hold tokens) and $D : T \rightarrow \mathbb{R}^+$ is a clock assignment function assigning nonnegative real valued ages to transitions. With every transition $t \in T$, its associated *preset* is $\bullet t = \{p \in P \mid (p, t) \in F\}$. The *postset* of a transition is defined as $t\bullet = \{p \in P \mid (t, p) \in F\}$. Note that the preset and postset for places are defined similarly. A transition, t, is *enabled* in a state if the members of its preset form a subset of the places in the marking of the state (i.e., $\bullet t \subseteq M$). A transition, t, is *fireable* in a state if it has been enabled longer than its lower timing bound (i.e., $D(t) \geq l(t)$). A transition, t, *must fire* before it has been enabled longer than its upper timing bound (i.e., $D(t) \leq u(t)$).

An example TPN is shown in Figure 1(a). In the initial state, transitions $a+$ and $b+/1$ are enabled, and exactly one of these transitions fires within 2 to 5 time units. The /1 and /2 notations in the TPN of Figure 1(a) indicate different transitions on the same signal wire. If $a+$ fires, then the $b+/2$ transition becomes enabled and fires within 2 to 5 more time units enabling $d+$.

2.2. Netlists

The goal of this work is to verify the correctness of a circuit implementation against a given TPN specification. The circuit to be verified is described using a netlist modeled by a directed graph $NET = \langle V, E \rangle$ where:

- $V = I \cup O \cup N$ is the set of vertices in the circuit;
- $E \subseteq (I \cup O \cup N) \times (N \cup O)$ is the set of edges.

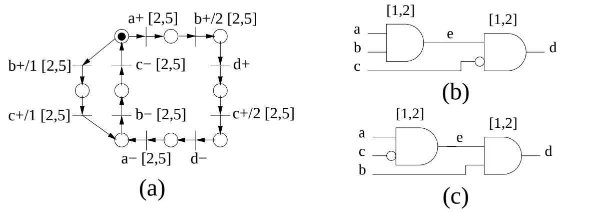

Figure 1: (a) TPN for our running example. (b) A netlist that is hazardous under the speed-independent model. (c) A netlist that is hazard-free under the speed-independent model.

Each vertex $v \in V$ represents a node in the netlist. This set is composed of both the input wires, I, and output wires, O, from the TPN description, as well as new nodes internal to the circuit, N. Each $e \in E$ represents a directed connection in the netlist from one node to another node. The set of *fanins* to a node is denoted by $FI(v)$, and the *fanouts* are denoted by $FO(v)$. Each node which is in $N \cup O$ has an associated gate output function $f_v(v_1, \ldots, v_r)$ where $FI(v) = \{v_1, \ldots, v_r\}$. This gate output function also has an associated minimum, min_v, and maximum, max_v, gate delay.

The netlist for a possible circuit implementation of the signal d in our example is shown in Figure 1(b). The set of vertices, V, is $\{a, b, c, d, e\}$, and the set of edges, E, is $\{\ (a, e), (b, e), (e, d), (c, d)\ \}$. The function associated with e is $f_e(a, b) = AND(a, b)$ which has a delay of 1 to 2 time units. An alternative circuit implementation for signal d is shown in Figure 1(c). The delays shown on the gates could just as easily be shown on the wires. As shown later, the primary concern here is with the maximum delay path from primary inputs to outputs.

The verification method described in this paper requires that the primary outputs must cut the circuit. In other words, if all primary outputs are removed from the netlist, the netlist would become acyclic. Intuitively, this means there can be no internal cycles in the netlist. Since the goal of this work is to use this verifier as a hazard checker during technology mapping and the technology mapper that has been developed satisfies this restriction, this seems acceptable. However, in the future, we are interested in generalizing this work to the case where there is internal feedback.

2.3. State graphs

In order to check correctness, a verification method typically uses a specification such as a TPN and a representation of the circuit implementation such as a netlist and finds all possible states represented using a state graph (SG). This verification method then checks the SG (often on the fly as the SG is being generated) for various correctness properties.

A SG is a labeled directed graph whose nodes are *states* and edges are *state transitions*. Formally, a SG is modeled by the tuple $SG = \langle\ S, T, \delta\ \rangle$ where:

- S is the set of states.
- T is the set of transitions from the TPN.
- $\delta \subseteq S \times T \times S$ is the set of state transitions.

Each individual state $s \in S$ is modeled as a tuple $s = \langle \nu, z \rangle$ where

- $\nu \subseteq V$ is the set of wires that are high in the state.
- z is a *zone* representing timing relationships.

Timing information is described using zones which are typically represented using *difference bound matrices* (DBMs) [4]. These matrices represent time differences between recently fired transitions. Each entry, z_{ij}, in the matrix represents a timing relationship of the form $\tau_{t_i} - \tau_{t_j} \leq z_{ij}$ where τ_{t_i} is the time at which t_i fires. In other words, z_{ij} represents the maximum amount of time in which t_i fires after t_j. An example zone for the point right after $a+$ fires is given below which represents the relationship $2 \leq \tau_{a+} - \tau_{c-} \leq 5$.

$$\begin{array}{c|cc|} & \tau_{c-} & \tau_{a+} \\ \tau_{c-} & 0 & -2 \\ \tau_{a+} & 5 & 0 \end{array}$$

Using a timed state space exploration algorithm such as the ones in [3, 10], it is possible to derive a SG using a TPN to drive the inputs and check the outputs, and a netlist to drive the outputs. However, the key result of this paper is that our method never explicitly derives this SG. Instead, a SG for a *complex gate equivalent* (CGE) version of the netlist needs to be derived. The CGE circuit for both netlists in Figures 1(b) and (c) is shown in Figure 2(a). The SG found using this circuit and the TPN in Figure 1(a) is shown in Figure 2(c). Each state vector ν is labeled in the state diagram to show the value of all signal wires and the zones are omitted for clarity. Each edge of the state graph is labeled with a signal transition $t \in T$. The input wire set is $\{a, b, c\}$ and the output wire set is $\{d\}$. There are nine states including 0000 and 1000, and ten state transitions including $(0000, a+, 1000)$. One last thing to note is that during the state space exploration to derive this SG, our method checks that the given CGE circuit is equivalent to the desired one. For example, if the CGE circuit given had been the one in Figure 2(b), after $a+$ fires, the netlist could produce a $d+$ when one is not expected in the TPN. This complex gate equivalence failure would then be reported to the user.

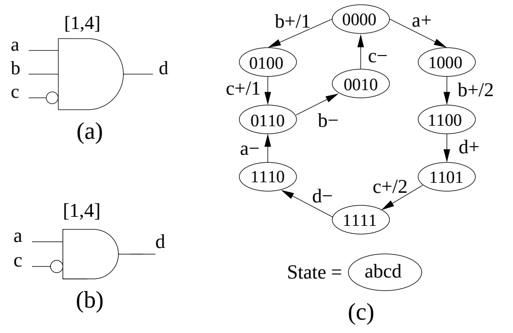

Figure 2: (a) Correct CGE circuit. (b) Incorrect CGE circuit. (c) State graph for correct CGE circuit.

3. VERIFICATION ALGORITHM

In [2, 1], the following theorem giving sufficient conditions for correctness of a speed-independent asynchronous circuit is presented (reworded to match the notation used in this paper). These conditions are also sufficient for correctness of timed circuits.

THEOREM 3.1. *(Sufficient conditions for correctness). Let NET* $= \langle V, E \rangle$ *be a circuit implementing the behavior described by TPN* $= \langle W, T, P, F, M_0, s_0, l, u, L \rangle$. *The NET is a correct implementation of the TPN if (1) it is complex gate equivalent to the TPN, and (2) it satisfies the acknowledgment and monotonicity properties.*

Our verification algorithm to check these correctness conditions is shown in Figure 3. This algorithm takes as input a TPN representing the possible input behavior and the required output behavior and a netlist, *NET*, representing the circuit to be checked. It then determines if the circuit is correct. When the circuit is not correct, this algorithm reports the locations of the errors that it finds. This algorithm is described in detail in this section.

```
verify(TPN, NET) {
  SG = check_equivalence(TPN, NET)
  find_stable_states(TPN, SG, NET)
  check_acknowledgment(SG, NET)
  check_monotonicity(SG, NET)
}
```

Figure 3: Top-level algorithm for verification.

3.1. Checking equivalence

The `check_equivalence` function forms a CGE netlist, uses this netlist and the given TPN to derive a SG, and checks if the CGE netlist provides outputs only at specified times.

The first step is to derive a CGE netlist in which there are no internal signals. In other words, it derives a netlist that has one gate per primary output signal. The Boolean function for this gate is specified only in terms of the primary inputs and outputs. The delay of this gate is set to the minimum and maximum delay from *any* input to the primary output. Although false paths through the logic may exist, our algorithm need not identify them at this point. Their inclusion results in a higher and thus more conservative maximum delay. At worst, this may result in a node being falsely determined to be hazardous.

In our example, the CGE representation for the netlists shown in Figures 1(b) and (c) is shown in Figure 2(a). The delay for this gate is set to $[1, 4]$, since in both cases there exists a minimum delay path of 1 time unit and a maximum delay path of 4.

Using this CGE netlist and the given TPN, a SG is found using a timed state space exploration algorithm. During the course of this state space exploration, output firings are checked. If an output fires prematurely, such as in the example shown in Figure 2(b), an error is reported to the user. Also, if an output is expected and the circuit does not provide one, an error is reported. In our example, if the function $f_d = AND(a, c)$ is used, after $a+$ and $b+$, a $d+$ would be expected, but the circuit would not produce it. This models a progress condition similar to *completeness with respect to specification* [6] and *strong conformance* [7]. When no errors are detected, `check_equivalence` returns a SG.

3.2. Finding stable states

After the `check_equivalence` step, our method has shown that the circuit is correct at a complex gate level. By hiding the internal signals before finding the state space, the state space is potentially reduced from $O(2^{|I|} * 2^{|O|} * 2^{|N|})$ to $O(2^{|I|} * 2^{|O|})$. When the number of internal signals is large, as is often the case in real designs, this savings can be quite dramatic. However, hazards on internal nodes can still produce incorrect circuit behavior. Therefore, it is now necessary to check that all internal nodes are hazard-free. This is accomplished by determining internal signal behavior implicitly. In particular, our method annotates each state with stability information about each internal signal. The goal of the `find_stable_states` algorithm is to determine in which states and for which state transitions in the complex gate SG that each internal node is stable. This is accomplished by deriving a predicate stable(s,n) for each state $s \in S$ and node $n \in N$ and another predicate stable(s,s',n) for each state transition $(s, t, s') \in \delta$. This stability information can then be used to determine if there are any hazards in the given netlist.

The algorithm to find the stability information is shown in Figure 4. The algorithm begins by first determining the predicate eval(s,n) by finding the Boolean evaluation in each state in the SG for each node in the netlist. This can be accomplished by simply fixing the values for each primary input and output in the netlist to the values given in the state and propagating this information through the netlist. From the SG in Figure 2(c) and netlist in Figure 1(b), eval(1100,e) and eval(1100,d) are determined to both be 1. For node e, the states in the set {1100, 1101, 1111, 1110} evaluate to 1 while the remaining states evaluate to 0.

```
find_stable_states(TPN,SG,NET) {
  foreach $s \in S$ and $n \in V$ find eval($s$,$n$)
  foreach $n \in N$, $s \in S$, and $(s,t,s') \in \delta$,
    stable($s$,$n$) = stable($s$,$s'$,$n$) = FALSE
  stabilize_timed(TPN,SG,NET)
  do
    distribute(SG)
    modified = stabilize_untimed(SG,NET)
  while modified
}
```

Figure 4: Algorithm for finding stable states.

The algorithm next initializes the stability predicates to FALSE to initially indicate that it is not known whether the internal signals are stable or changing. The goal of the rest of the algorithm is to determine stability of the internal signals, whenever possible. In the next subsection, a brief review of untimed stabilization is given which comes from the work in [2]. In the following subsection, we discuss our new contribution which is timed stabilization. The timed stabilization routine does not need to be iterated, so it is executed first. The untimed stabilization routine may require iteration since stabilizations on one node of the network can influence stabilizations on other nodes.

3.2.1. Untimed stabilization

The objective of stabilization is to show that at some points in the SG, the evaluations of some internal node, n, are certain to be stable. The algorithm to determine untimed stability is shown in Figure 5. An internal node is considered untimed stable if a change in evaluation on an internal node is acknowledged on a primary output. In other words, for a state transition (s, t, s'), if the transition t could only have occurred if the internal node n is stable at its Boolean evaluation, then it can be said that the transition t has acknowledged that the node n is stable.

To determine if an internal node n is acknowledged to be stable by a state transition (s, t, s'), it must first be determined if a path exists from n to the output transition under consideration. It must

```
stabilize_untimed(SG,NET) {
  modified = FALSE
  foreach $n \in N$
    foreach $(s,t,s') \in \delta$
      if ((exists_path(NET,$n$,$L(t)$)) and
          (must_prop(NET,$s$,$n$,$L(t)$)) and
          (not stable($s$,$s'$,$n$))) then
        stable($s$,$s'$,$n$) = TRUE
        modified = TRUE
  return modified
}
```

Figure 5: Untimed stabilization algorithm.

then be determined using the function must_prop if the value at n must propagate through any possible path to the output. This is done by ensuring that all functions in the path from n to the output have non-controlling values on the side inputs. Consider the example netlist in Figure 1(b) and the state transition (1100,$d+$,1101). There exists a path between node e and the output d. In state 1100, node e evaluates to 1. This value at e must propagate to the output because d cannot go high until e has gone high. More succinctly, output d switched from low to high as a direct consequence of node e going high and the side input, c being at 0. Therefore, stable(1100,1101,e) is set to TRUE.

Next, the distribute function is used to copy this stabilization forward in the state graph until a change in evaluation is encountered. In particular, stable(1100, 1101, e) implies the following stability conditions are TRUE: stable(1101,e), stable(1101,1111,e), stable(1111,e), stable(1111,1110, e), stable(1110,e), and stable(1110,0110,e). This distribution of stability information halts when it reaches state 0110 since the Boolean evaluation of e in this state changes from 1 to 0.

The other transition in the SG that could possibly indicate an untimed stabilization for node e is the state transition (1111, $d-$, 1110). In this case, however, the input c is 1 (a controlling value), prohibiting the propagation of node e to the output d. Thus, no stabilization can be assumed for the falling transition of e. As explained later, this lack of stabilization on the falling transition of e indicates a hazard on node e.

A similar analysis done on the circuit in Figure 1(c) shows that the rising transition on node e is acknowledged by $d+$ and the falling transition is acknowledged by $d-$ since b is high (a non-controlling value) when d goes low. As a result, this circuit can be shown to be hazard-free under the speed-independent model.

3.2.2. Timed stabilization

When timing information is taken into account, the hazard found for the netlist shown in Figure 1(b) may not actually manifest. If this is the case, then node e is hazard-free. This subsection describes our new method to determine stabilization using timing information. Timed stabilization attempts to show further stability in the state graph by calculating the maximum possible time through the network to the node of interest, n, and comparing this against the minimum time spent traversing the state graph. When it can be shown that in the worst-case a sufficient amount of time has elapsed, node n can be stabilized.

The algorithm to determine timed stabilization is shown in Figure 6. For each node n, the algorithm first measures the longest path delay from any primary input or output to the node n. This must be done because the actual signal that causes n to change evaluation may not be known due to differences in path lengths. For our example netlist in Figure 1(b), this delay is determined to be 2. Next, the algorithm initializes the visit array which is used to let the recursion know when a state has been visited along multiple paths when determining stabilization of node n. At this point, the algorithm finds state transitions, (s, t_i, s'), where the Boolean evaluation of n changes. This indicates locations in the state graph where the node n becomes unstable. The algorithm then takes the zone z associated with state s and updates it to include the transition t_i. The reason this is done rather than taking the zone associated with s' is that t_i may have been pruned from this zone. It is important that t_i is in the zone that is used for timed stabilization as t_i serves as a reference transition as the algorithm moves forward in the state graph. Finally, the algorithm initializes a path array which is used to terminate cycles during the analysis of a path in the SG.

```
stabilize_timed(TPN,SG,NET) {
  foreach $n \in N$
    d = find_max_delay(NET,$n$)
    foreach $s \in S$
      visit($s$) = FALSE
    foreach ($s$,$t_i$,$s'$) $\in \delta$ where $s$=$\langle v,z \rangle$
      if (eval($s$,$n$) $\neq$ eval($s'$,$n$)) then
        $z'$ = update_zone(TPN,NET,$z$,$t_i$)
        foreach $s'' \in S$
          path($s''$)=FALSE
        do_timed(TPN,SG,NET,$n$,$s'$,$z'$,$t_i$,
                 d,visit,path)
}
```

Figure 6: Timed stabilization algorithm.

The update_zone algorithm shown in Figure 7 adds a new transition to a given zone. The first step is to extend the zone to include a new row and column for the new transition, t_i. Next, it searches the zone starting with the transitions that have been added most recently for transitions that enable t_i (i.e., $t_j \in \bullet\bullet t_i$). The first such transition that it finds is the causal transition for t_i. The upper bound of the firing time for t_i should be set in reference to this transition. The upper bound is either taken from the TPN when t_i is a transition on an input wire or it is taken as the maximum delay in the netlist generating t_i when it is a transition on an output wire. For all transitions that enable t_i, a lower bound must be set between t_j and t_i. For all transitions that do not enable t_i, the timing relationships are initially set to be unbounded. At this point, the zone is recanonicalized using Floyd's all-pairs shortest path algorithm to tighten any loose inequalities. This recanonicalization step is necessary because tightened bounds increase accuracy. In addition, there are often cases where no timing relationship is known between a newly entered transition and the other entries in the zone. Recanonicalization creates these entries in the zone. As an example, the zone found for the state 1110 in our example is shown in Figure 8(a). The new zone after adding the transition $a-$ is shown in Figure 8(b).

The do_timed algorithm shown in Figure 9 is used to recursively explore the SG, attempting to accumulate sufficient time to stabilize a given node n before reaching a termination condition. This algorithm first marks the current state s as visited in the visit and path arrays described earlier. Next, it considers each state transition (s, t_k, s'). First, it adds the transition, t_k, to the zone. Next, it checks the zone to determine if enough time has

```
update_zone(TPN,NET,$z$,$t_i$) {
  $z'$ = extend($z$,$t_i$)
  found_causal = FALSE
  foreach $t_j \in z'$ in reverse order
    if ($t_j \in \bullet\bullet t_i$) then
      if (!found_causal) then
        found_causal = TRUE
        if ($L(t_i) \in I$) then
          $z'_{ij} = u(t_i)$
        else
          $z'_{ij}$ = find_max_delay(NET,$L(t_i)$)
      if ($L(t_i) \in I$) then
        $z'_{ji} = (-1) * l(t_i)$
      else
        $z'_{ji} = (-1) *$ find_min_delay(NET,$L(t_i)$)
    else
      $z'_{ij} = \infty$
      $z'_{ji} = \infty$
  recanonicalize($z'$)
}
```

Figure 7: Algorithm to update the zone.

$$\begin{array}{c|c|}
 & \tau_{d-} \\
\tau_{d-} & 0
\end{array}$$

(a)

$$\begin{array}{c|cc|}
 & \tau_{d-} & \tau_{a-} \\
\tau_{d-} & 0 & -2 \\
\tau_{a-} & 5 & 0
\end{array}$$

(b)

$$\begin{array}{c|ccc|}
 & \tau_{d-} & \tau_{a-} & \tau_{b-} \\
\tau_{d-} & 0 & -2 & \infty \\
\tau_{a-} & 5 & 0 & -2 \\
\tau_{b-} & \infty & 5 & 0
\end{array}$$

(c)

$$\begin{array}{c|ccc|}
 & \tau_{d-} & \tau_{a-} & \tau_{b-} \\
\tau_{d-} & 0 & -2 & -4 \\
\tau_{a-} & 5 & 0 & -2 \\
\tau_{b-} & 10 & 5 & 0
\end{array}$$

(d)

$$\begin{array}{c|cccc|}
 & \tau_{d-} & \tau_{a-} & \tau_{b-} & \tau_{c-} \\
\tau_{d-} & 0 & -2 & -4 & \infty \\
\tau_{a-} & 5 & 0 & -2 & \infty \\
\tau_{b-} & 10 & 5 & 0 & -2 \\
\tau_{c-} & \infty & \infty & 5 & 0
\end{array}$$

(e)

$$\begin{array}{c|cccc|}
 & \tau_{d-} & \tau_{a-} & \tau_{b-} & \tau_{c-} \\
\tau_{d-} & 0 & -2 & -4 & -6 \\
\tau_{a-} & 5 & 0 & -2 & -4 \\
\tau_{b-} & 10 & 5 & 0 & -2 \\
\tau_{c-} & 15 & 10 & 5 & 0
\end{array}$$

(f)

Figure 8: Zone creation and evolution.

accumulated from the reference transition t_i to the new transition t_k such that the node of interest n has certainly stabilized. If it has, it must also check that the state s' has not been visited along a different path. It must be the case that the minimum time upon reaching a state along all paths to that state has exceeded the maximum logic delay d. Therefore, if this state is encountered along a different path and did not stabilize, then this state transition cannot stabilize the node n. If the amount of accumulated delay does not exceed the delay d, then the algorithm must determine if it is going to recurse down this state transition. If this state has been seen previously upon this path, the algorithm has encountered a cycle of states and must not recurse. If the Boolean evaluation of the node n has changed, then again the algorithm must not recurse. If this is a new state on this path and the Boolean evaluation is maintained, then the algorithm recursively visits the state s'. Note that this edge may have been found to be stable along a different path, but it is not stable along the path the algorithm is currently working on. Therefore, the algorithm must say this edge is not stable before recursing. Upon returning from recursion, the path variable is set to false to allow other potential paths to visit the state s.

```
do_timed(TPN, SG, NET, $n$, $s$, $z$, $t_i$, d, visit, path) {
  visit($s$) = path($s$) = TRUE
  foreach ($s$, $t_k$, $s'$) in $\delta$
    $z'$ = update_zone(TPN, NET, $z$, $t_k$)
    if (-1*$z'_{ik}$ > d) then
      if (not visit($s'$)) then
        stable($s$, $s'$, $n$) = TRUE
    else if (not path($s'$) and
          eval($s$, $n$) == eval($s'$, $n$)) then
      stable($s$, $s'$, $n$) = FALSE
      do_timed(TPN, SG, NET, $n$, $s'$, $z'$, $t_i$,
               d, visit, path)
  path($s$) = FALSE
}
```

Figure 9: Timed stabilization recursion.

Our algorithm has the potential for requiring the exploration of a large number of paths, although experimental results have not shown this to happen. The further the algorithm recurses through the state graph, the more potential side paths there are to explore. Typically, the length of the paths explored is very short as the recursion terminates quickly. If in the future, examples are found where this is not the case, the algorithm can be changed to limit the path length. This can improve efficiency at the potential cost of more false negative results.

Let us consider again the example netlist in Figure 1(b). A change in evaluation on node e occurs between states 1110 and 0110. As mentioned previously, the do_timed function is called with the zone shown in Figure 8(b). As the SG is traversed, the next transition encountered is $b-$. Since $b-$ fires 2 to 5 time units after $a-$, these entries are entered into the appropriate rows and columns as shown in Figure 8(c). The timing of the other non-diagonal entries are set to ∞. The zone is then recanonicalized and the resulting zone is shown in Figure 8(d). The parameter of interest is the minimum elapsed time between the last transition entered, $b-$, and the initial transition $a-$ which is 2 in this case. Note that lower bounds appear as negative values in a DBM. Since two time units is insufficient time to say with certainty that node e has stabilized, the algorithm considers recursing on state 0010. Since this state has not yet been explored on this path, and since node e still evaluates to 0 in this state, the algorithm recurses to state 0010. Upon recursion, the algorithm adds transition $c-$ to the zone as shown in Figure 8(e) and recanonicalizes to obtain the zone shown in Figure 8(f). The new minimum time elapsed from $a-$ till $c-$ is 4 time units. Since this number is larger than the maximum delay of the AND gate (2 time units), the algorithm can mark this edge as stabilized. The distribute function then copies this stabilization onto states 0000, 0100, and 1000 and edges $(0000, b+/1, 0100)$, $(0100, c+/1, 0110)$, $(0000, a+, 1000)$, and $(1000, b+/2, 1100)$. This is significant in that the hazard condition that existed after untimed stabilization cannot manifest because of the timing relationships between the circuit and the SG.

3.3. Checking for hazard-freedom

Hazards can manifest in asynchronous circuits due to violations in the *acknowledgment* or *monotonicity* properties [2]. This section gives an explanation of how our method checks for violations in acknowledgment and monotonicity. This explanation though is brief since it is essentially the same as that in [2].

An acknowledgment violation occurs when an internal node becomes excited to change to a new value, but its excitation changes value before it can be shown to have stabilized. The algorithm shown in Figure 10 uses the stability information found earlier to check for acknowledgment on all excited nodes. The algorithm examines each node n and each state transition (s, t, s') in which n changes Boolean evaluation. If n is not stable before it changes Boolean evaluation, then an acknowledgment hazard is reported. For the netlist shown in Figure 1(b) using only untimed stabilization, a hazard is found on node e for the state transition $(1000, b+/2, 1100)$. However, since timed stabilization detects that this edge is stable for node e, this circuit is hazard-free when timing is considered.

```
check_acknowledgment(SG, NET) {
  foreach ($n \in N$)
    foreach $(s, t, s') \in \delta$
      if ((eval($s$, $n$) $\neq$ eval($s'$, $n$)) and
        not stable($s$, $s'$, $n$)) then
        report hazard for ($s$, $s'$, $n$)
}
```

Figure 10: Algorithm to check acknowledgment.

A monotonicity violation occurs when an internal or output node is supposed to remain stable but it becomes momentarily excited or when it is supposed to make a transition but it makes the transition non-monotonically. This occurs when a gate has a *potential hazard* while there is no stable, *forcing* side-input. For example, a potential hazard exists when the output of an AND gate is supposed to remain stable low or fall, but one input is rising. If a side-input cannot be found that is stable low while the other input is rising, it is possible that the AND gate may momentarily evaluate to 1 causing a glitch on the output of the AND gate. The algorithm to check for monotonicity violations is given in Figure 11. The definitions of potential_hazard and forcing are a bit involved and can be found in [2].

```
check_monotonicity(SG, NET) {
  foreach ($n \in N$)
    foreach ($s \in S$)
      foreach $v \in$ FI($n$)
        if (potential_hazard($s$, $n$, $v$)) then
          if ($\neg\exists\ w \in W$ s.t. forcing($s$, $n$, $w$)
            and stable($s$, $w$)) then
            report hazard on $n$ for ($s$, $v$)
}
```

Figure 11: Algorithm to check monotonicity.

4. EXPERIMENTAL RESULTS

The gate-level timing verification method described in this paper has been implemented and tested on numerous examples. Table 1 compares our new gate-level timing verification method using standard benchmarks against results for the timed automata tool KRONOS [9], a conservative approximation method described in [14], and the ATACS explicit state timing verifier [12]. For KRONOS runtimes, an entry with a question mark indicates the amount of time after which the verification ran out of memory. The runtimes for KRONOS and Pena's methods are taken from their papers while the runtimes for ATACS and our new method are from a 900 MHz Pentium 4 with 256MB of memory. For our new method,

Table 1: Comparison of standard benchmarks against other timing verification tools.

		KRONOS	PENA	ATACS		New Method		
Example	Gates	CPU Time(s)	CPU Time(s)	CPU Time(s)	Mem(MB)	CPU Time(s)	Mem(MB)	Hazards
alloc-outbound	11	0.09	3	0.33	5.6	0.09	2.9	0/0
chu133	9	0.63	1	0.16	3.0	0.11	2.2	1/1
converta	12	0.19	12	0.24	3.8	0.11	1.8	2/2
dff	6	0.19	3	0.12	2.5	n/a	n/a	3/?
ebergen	9	0.14	1	0.15	3.0	0.13	1.8	3/3
half	7	0.41	1	0.13	2.2	0.08	1.5	1/1
mp-forward-pkt	10	0.24	5	0.17	3.5	0.10	2.5	0/0
nowick	10	0.05	3	0.20	3.8	0.10	2.0	0/0
rcv-setup	6	0.22	1	0.16	3.2	0.08	1.8	0/0
rpdft	8	2.93	2	0.30	4.0	0.10	1.9	1/2
sbuf-ram-write	17	31.77	415	0.32	5.8	0.20	3.7	1/2
sbuf-read-ctl	10	0.13	2	0.14	3.3	0.10	2.5	0/0
sbuf-send-ctl	13	54	0.49	0.65	6.1	0.10	2.8	1/1
sbuf-send-pkt2	13	0.07	103	0.42	6.6	0.10	3.1	0/1
vme	12	0.39	30	0.39	4.9	n/a	n/a	1/?
mr1	16	607.43	317	0.30	5.1	n/a	n/a	0/?
tsend-bm	12	589.56	46	5.32	8.6	n/a	n/a	1/?
mmu	22	595.09?	480	0.53	7.1	n/a	n/a	0/?
mr0	20	593.24?	48	0.55	7.1	n/a	n/a	0/?
ram-read-sbuf	17	678.48?	550	0.34	6.0	0.18	3.4	0/0
trimos-send	24	580.33?	127	10.7	25.0	4.87	3.6	5/5

an entry of n/a indicates that this example has an internal cycle and cannot be analyzed using our new method. For the smaller examples, our method has comparable and usually better runtimes than the other methods. However, for larger examples with more concurrency such as trimos-send, our method is more than two orders of magnitude faster than KRONOS, twenty-five times faster than Pena's tool, and twice as fast as the explicit state method in ATACS. In addition, our new method shows some reduction in memory usage as compared to the ATACS explicit state timing verifier. This reduction in run-time and memory usage is directly related to the reduced complexity of the SG as stated earlier.

Since our goal is to determine which gates have hazards on their outputs, the explicit method in ATACS is configured to continue after finding one hazard and identify all hazards. It should be noted that KRONOS did not check for hazards, but instead was only checking conformance while Pena's tool halts after a hazard is found. The last column of the table indicates the number of gates that have hazards found by the explicit state method and our new method. Despite being a conservative approximation, our method found the exact number of hazards in most cases. However, in three examples, *rpdft*, *sbuf-ram-write*, and *sbuf-send-pkt2*, our new method found one additional false hazard.

The key advantage of our new method is its ability to be able to efficiently verify circuits with a large number of internal signals. In order to demonstrate this, a few of our benchmark circuits derived from a variety of sources were selected, and gate-level circuits were derived for them that use only 2-input NAND gates and inverters. Our results are shown in Table 2. In all the examples, our method is still able to check for hazards in 1.2 seconds or less while for the largest examples the explicit state method cannot complete.

5. CONCLUSIONS

This paper presents a new method for efficiently checking hazard-freedom in gate-level timed circuits. This method uses a cube approximation of the internal signal behavior in order to avoid generating an explicit state graph representing the switching behavior of the internal signals. Our experimental results show that this new method can be substantially faster than previous gate-level timing verification tools. While this method is conservative and thus can report some incorrect hazards, the number of such false negative results appears to be small. This method has been shown to scale very well in that it can verify examples with more than 150 gates in less than a second while previous methods fail to complete.

In the future, we plan to develop techniques to evaluate if a hazard is false or not. When an acknowledgment hazard is found on a node n, the state transition, (s,t,s'), where the hazard occurs is reported. For monotonicity hazards, the state s and input v that cause the monotonicity violation are reported. In either case, this information can be used to create an error trace from the initial state. This error trace can then be used to perform a guided simulation of the circuit to detect if the hazard can occur or not. While in theory, this simulation could result in a full state space exploration, it is likely only to require exploration of a small subset of the state space to determine if it is false or not.

In the future, we also plan to utilize this hazard analyzer to implement a technology mapper for timed circuits. In asynchronous circuits, hazards must be avoided and care must be taken during technology mapping to not introduce hazards in the design. Therefore, an asynchronous technology mapper requires a method to rapidly determine when a transformation of the netlist has introduced a hazard. The hazard analyzer described in this paper addresses this need making efficient technology mapping of timed circuits possible.

Table 2: Comparison for decomposed netlists.

Example	Gates	ATACS CPU Time(s)	ATACS Mem(MB)	New Method CPU Time(s)	New Method Mem(MB)	Hazards
scsiSV	18	1.35	7.9	0.13	1.3	0/0
slatch	29	33.5	53.4	0.15	1.8	0/0
lapbsv	37	20.0	41.5	0.17	1.3	0/0
elatch	38	183	229	0.28	1.8	0/0
cnt3	80	>1000	>256	0.24	1.7	?/15
srgate	85	>1000	>256	0.29	2.3	?/0
selopt	164	>2000	>256	0.90	3.3	?/46
cnt11	213	>2000	>256	1.20	4.8	?/78

6. REFERENCES

[1] P. A. Beerel, J. R. Burch, and T. H.-Y. Meng. Checking combinational equivalence of speed-independent circuits. *Formal Methods in System Design*, Mar. 1998.

[2] P. A. Beerel, T. H.-Y. Meng, and J. Burch. Efficient verification of determinate speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 261–267. IEEE Computer Society Press, Nov. 1993.

[3] W. Belluomini and C. J. Myers. Timed state space exploration using posets. *IEEE Transactions on Computer-Aided Design*, 19(5):501–520, May 2000.

[4] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, 1989.

[5] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[6] J. Ebergen and S. Gingras. A verifier for network decompositions of command-based specifications. In *Proc. Hawaii International Conf. System Sciences*, volume I. IEEE Computer Society Press, Jan. 1993.

[7] G. Gopalakrishnan, E. Brunvand, N. Michell, and S. Nowick. A correctness criterion for asynchronous circuit validation and optimization. *IEEE Transactions on Computer-Aided Design*, 13(11):1309–1318, Nov. 1994.

[8] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. Analysis and identification of speed-independent circuits on an event model. *Formal Methods in System Design*, 4(1):33–75, 1994.

[9] O. M. M. Bozga, H. Jianmin and S. Yovine. Verification of asynchronous circuits using timed automata. In O. M. Eugene Asarin and S. Yovine, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[10] E. Mercer, C. Myers, and T. Yoneda. Improved poset timing analysis in timed petri nets. In *The Tenth Workshop on Synthesis and System Integration of MIxed Technologies (SASIMI 2001)*, October 2001.

[11] P. Merlin and D. J. Faber. Recoverability of communication protocols. *IEEE Trans. on Communication*, COM-24(9):1036–1043, 1976.

[12] C. J. Myers, W. Belluomini, K. Killpack, E. Mercer, E. Peskin, and H. Zheng. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 335–340, Feb. 2001.

[13] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng. POSET timing and its application to the synthesis and verification of gate-level timed circuits. *IEEE Transactions on Computer-Aided Design*, 18(6):769–786, June 1999.

[14] M. A. Peña, J. Cortadella, A. Kondratyev, and E. 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, Apr. 2000.

[15] O. Roig, J. Cortadella, and E. Pastor. Hierarchical gate-level verification of speed-independent circuits. In *Asynchronous Design Methodologies*, pages 129–137. IEEE Computer Society Press, May 1995.

[16] O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *16th International Conference on the Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 374–391, June 1995.

[17] K. S. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. J. Myers, K. Y. Yun, R. Koi, C. Dike, and M. Roncken. An asynchronous instruction length decoder. *IEEE Journal of Solid-State Circuits*, 36(2):217–228, Feb. 2001.

[18] T. Yoneda and H. 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.

[19] H. Zheng, E. Mercer, and C. Myers. Modular verification of timed circuits using automatic abstraction. *IEEE Transactions on Computer-Aided Design*, 22(9), Sept. 2003.

[20] H. Zheng, C. J. Myers, D. Walter, S. Little, and T. Yoneda. Verification of timed circuits with failure directed abstractions. In *Proc. International Conf. Computer Design (ICCD)*. IEEE Computer Society Press, Oct. 2003.