

# Efficient Verification of Hazard-Freedom in Gate-Level Timed Asynchronous Circuits

Curtis A. Nelson  
University of Utah

Chris J. Myers  
University of Utah

Tomohiro Yoneda  
National Institute of  
Informatics, Japan

## 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, each internal node and output of the circuit must be verified for hazard-freedom. 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. 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 can potentially reduce the amount of circuitry that would be needed for 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 which is three times faster while using half the power of the comparable synchronous design [9].

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. 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 [2, 7, 4] 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 have been many methods developed for the verification of gate-level speed-independent asynchronous circuits [1, 8]. In speed-independent circuits, no timing assumptions are made about gates or the environment. In [1], an efficient verification method for determinate speed-independent circuits is proposed. This work reduces the state space explosion problem found in earlier methods 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 [1].

The goal of this paper is to extend the work in [1] to perform efficient verification of timed asynchronous 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 we can track the time evolution in the state space, then it may be possible to identify the stability of internal signals. Using this *timed cube approximation*, we can rapidly analyze a gate-level timed circuit for hazards.

## 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 specified environment, and a *netlist* representing the circuit to be verified. The verification procedure creates and uses a *state graph* (SG) to represent reachable timed states. This section describes each of these formally.

### 2.1 Time Petri nets

Our method uses TPN's to model the possible input behaviors and the required output behaviors for timed circuits [5]. Let  $W$  be a finite set of wires in a timed circuit. 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 digraph 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 non-negative 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 in a similar manner. 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.,  $D(t)$  must never exceed  $u(t)$ ). An example TPN is shown in Figure 1(a) where the associated timing bounds  $[l, u]$  are shown for each transition.

## 2.2 Netlist

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 and  $E \subseteq (I \cup O \cup N) \times (N \cup O)$  is the edges between vertices.

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.

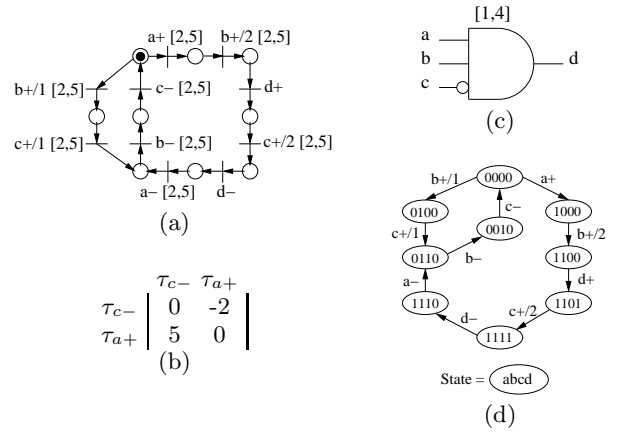
## 2.3 State graphs

A SG is a labeled directed graph whose nodes are *states* and edges are *state transitions*. Formally, a SG is modeled by the tuple  $SG = \langle S, \delta \rangle$  where  $S$  is the set of states and  $\delta \subseteq S \times T \times S$  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 and  $z$  is a *zone* representing timing relationships. Timing information in each state is given using zones which are typically represented using difference bound matrices (DBMs) [3]. 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  $\tau_{t_i}$  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 is shown in Figure 1b which represents the relationship  $2 \leq \tau_{a+} - \tau_{c-} \leq 5$ .

## 3. VERIFICATION ALGORITHM

Using a timed state space exploration algorithm such as the one in [2], it is possible at this point to derive a SG using a TPN to drive the inputs and check the outputs, and



**Figure 1: (a) Example TPN. (b) Example zone. (c) Complex gate equivalent circuit. (d) State graph for complex gate circuit.**

a netlist to drive the outputs. However, the key result of this paper is that our method never explicitly derives this SG. Instead, we only need to derive a SG for a *complex gate equivalent* (CGE) version of the netlist. A CGE circuit example is shown in Figure 1(c). The SG found using this circuit and the TPN in Figure 1(a) is shown in Figure 1(d).

Our verification algorithm to ensure correct operation of the CGE is shown in Figure 2. 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, and it determines if the circuit is correct.

```

verify( $TPN, NET$ )
   $SG = \text{check\_equivalence}(TPN, NET)$ 
  find\_stable\_states( $TPN, SG, NET$ )
  check\_acknowledgment( $SG, NET$ )
  check\_monotonicity( $SG, NET$ )

```

**Figure 2: Top-level algorithm for verification.**

The **check\\_equivalence** function forms a CGE netlist, uses this netlist and the given TPN to derive a SG, and checks if the complex gate netlist provides outputs consistent with the TPN specifications. If no errors are detected, **check\\_equivalence** returns a SG.

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, we reduce the potential size of the state space from  $O(2^{|I|} * 2^{|O|} * 2^{|N|})$  to  $O(2^{|I|} * 2^{|O|})$  representing an exponential reduction in the potential size of the state space. 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 3. The algorithm begins by first determining the predicate  $\text{eval}(s,v)$  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.

```

find_stable_states( $TPN, SG, NET$ )
  find  $\text{eval}(s,v)$  for all  $s \in S$  and  $v \in V$ 
  initialize  $\text{stable}(s,n)$  and  $\text{stable}(s,s',n)$  to FALSE
  for all  $n \in N$ ,  $s \in S$ , and  $(s,t,s') \in \delta$ 
  stabilize_timed( $TPN, SG, NET$ )
  do {
    propagate( $SG$ )
    modified = stabilize_untimed( $SG, NET$ )
  } while modified

```

**Figure 3: Algorithm for finding stable states.**

The algorithm next initializes the stability predicates to FALSE to indicate that initially we do not know whether the internal signals are stable or changing in each state and during each state transition. The goal of the rest of the algorithm is to determine stability of the internal signals, whenever possible. 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.

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. An internal node is considered untimed stable if a change in evaluation on the 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 we can say that the transition  $t$  has acknowledged that the node  $n$  is stable. Detailed algorithms for finding untimed stability can be found in [1].

Our new contribution to verifying hazard-freedom is extending stabilization by taking into account timing information. When timing information is used, nodes found to be hazardous in the untimed sense may not actually manifest. 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 and comparing 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 4. For each node  $n$ , the algorithm first measures the longest path delay from any primary input or output to the node  $n$ .<sup>1</sup> Next, the algorithm initializes the  $\text{visit}$  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 transi-

<sup>1</sup>In order to be safe, the absolute longest path is used and not just the longest path from  $L(t_i)$ . This must be done because the actual signal that causes  $n$  to change evaluation may not be  $L(t_i)$  due to differences in path lengths.

tion  $t_i$ . It is important that  $t_i$  is in the zone that we use for timed stabilization as  $t_i$  serves as a reference transition as we move forward in the state graph. Finally, the algorithm initializes a  $\text{path}$  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_maximum_delay( $NET, n$ )
    initialize  $\text{visit}(s)$  for all  $s \in S$ 
    foreach ( $(s,t_i,s') \in \delta$  where  $s=(v,z)$ )
      if ( $\text{eval}(s,n) \neq \text{eval}(s',n)$ ) then
         $z' = \text{update\_zone}(TPN, NET, z, t_i)$ 
        initialize  $\text{path}(s'')$  for all  $s'' \in S$ 
        do_timed( $TPN, SG, NET, n, s', z', t_i, d, \text{visit}, \text{path}$ )

```

**Figure 4: Timed stabilization algorithm.**

The  $\text{do\_timed}$  algorithm shown in Figure 5 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  $\text{visit}$  and  $\text{path}$  arrays described earlier. Next, it considers each state transition  $(s,t_j,s')$ . First, it adds the transition,  $t_j$ , to the zone. Next, it checks the zone to determine if enough time has accumulated from the reference transition  $t_i$  to the new transition  $t_j$  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 along a different path we encounter this state 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.

```

do_timed( $TPN, SG, NET, n, s, z, t_i, d, \text{visit}, \text{path}$ )
   $\text{visit}(s) = \text{TRUE}$ 
   $\text{path}(s) = \text{TRUE}$ 
  foreach ( $(s,t_j,s')$  in  $\delta$ )
     $z' = \text{update\_zone}(TPN, NET, z, t_j)$ 
    if ( $-1 * z'_{ij} > d$ ) then
      if (not  $\text{visit}(s')$ ) then
         $\text{stable}(s,s',n) = \text{TRUE}$ 
      else if (not  $\text{path}(s')$  &&
         $\text{eval}(s,n) == \text{eval}(s',n)$ ) then
         $\text{stable}(s,s',n) = \text{FALSE}$ 
        do_timed( $TPN, SG, NET, n, s', z', t_i, d, \text{visit}, \text{path}$ )
     $\text{path}(s) = \text{FALSE}$ 

```

**Figure 5: Timed stabilization recursion.**

Hazards can manifest in asynchronous circuits due to violations in the *acknowledgment* or *monotonicity* properties

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

Example	Gates	KRONOS CPU Time	PENA CPU Time	ATACS CPU Time	New Method CPU Time	Hazards
alloc-outbound	11	0.09	3	0.37	0.14	0/0
chu133	9	0.63	1	0.14	0.11	1/1
converta	12	0.19	12	0.26	0.09	2/2
ebergen	9	0.14	1	0.18	0.11	3/3
half	7	0.41	1	0.11	0.09	1/1
nowick	10	0.05	3	0.22	0.11	0/0
rpdf	8	2.93	2	0.35	0.11	1/2
sbuf-ram-write	17	31.77	415	0.40	0.18	1/2
sbuf-read-ctl	10	0.13	2	0.17	0.12	0/0
sbuf-send-ctl	13	54	0.49	0.87	0.14	1/1
sbuf-send-pkt2	13	0.07	103	0.61	0.16	0/1
ram-read-sbuf	17	678.48?	550	0.41	0.18	0/0
trimos-send	24	580.33?	127	13.75	5.06	5/5

[1]. 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. A monotonicity violation occurs when an internal or output node is supposed to remain stable but it becomes momentarily excited or it is supposed to make a transition which it makes non-monotonically.

#### 4. RESULTS AND CONCLUSIONS

Table 1 compares our new gate-level timing verification method using standard benchmarks against results for the timed automata tool KRONOS [4], a conservative approximation method described in [7], and the ATACS explicit state timing verifier [6]. For the 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 650 MHz Pentium III.

Since our goal is to determine which gates have hazards on their outputs, we configured the explicit method in ATACS to continue after finding one hazard and identify all hazards. It should be noted that KRONOS did not check for hazards, but it instead was only checking conformance while Pena’s tool halted after a hazard was 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 of many examples (not all shown), *rpdf*, *sbuf-ram-write*, and *sbuf-send-pkt2*, our new method found one additional false hazard.

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.

#### 5. REFERENCES

- [1] 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.
- [2] W. Belluomini and C. J. Myers. Timed state space exploration using posets. *IEEE Transactions on Computer-Aided Design*, 19(5):501–520, May 2000.
- [3] 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.
- [4] 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.
- [5] P. Merlin and D. J. Faber. Recoverability of communication protocols. *IEEE Trans. on Communication*, COM-24(9):1036–1043, 1976.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.