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

Curtis A. Nelson, *Member, IEEE*, Chris J. Myers, *Senior Member, IEEE*, and Tomohiro Yoneda, *Member, IEEE*

***Abstract*—This paper presents an efficient method for verifying hazard-freedom in gate-level timed asynchronous circuits. Timed circuits are a class of asynchronous circuits that are optimized using explicit timing information. 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 circuits require an explicit state exploration that often results in state explosion for even modest-sized examples. The goal of this paper 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 that could not be previously analyzed, in that it is capable of analyzing these circuits in less than a second. While this method is conservative in that some false hazards may be reported, our results indicate that their number is small.**

***Index Terms*—Hazard-freedom, technology mapping, timed asynchronous circuits, verification.**

I. Introduction

TIMED circuits are a class of asynchronous circuits that use explicit timing information during circuit synthesis. This timing information, however rough the estimates may be, can potentially reduce the amount of circuitry as compared with 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. 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 [1].

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] 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 [8]–[14]. 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 [10]. Determinate speed-independent circuits allow input choice (conditionals) but not output choice (arbitration). The work in [10] 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 [10].

Abstraction of internal nodes to combat state explosion is performed in [15] and [16]. 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. This work could potentially be used to verify hazard-freedom, so it may be interesting in the future to explore combining this approach with the one proposed in this paper.

This paper extends the work in [10] 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.

Manuscript received December 21, 2005; revised April 19, 2006. The work of C. A. Nelson and C. J. Myers was supported in part by the SRC under Contract 2002-TJ-1024 and Contract 2005-TJ-1357, in part by an SRC Graduate Fellowship, and in part by NSF Japan Program Award INT-0087281. This paper was recommended by Associate Editor R. F. Damiano.

C. A. Nelson is with the School of Engineering, Walla Walla College, College Place, WA 99342 USA (e-mail: nelscu@wwc.edu).

C. J. Myers is with the Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT 84112 USA (e-mail: myers@ece.utah.edu).

T. Yoneda is with the National Institute of Informatics, Tokyo 101-8430, Japan (e-mail: yoneda@nii.ac.jp).

Digital Object Identifier 10.1109/TCAD.2006.883912

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.

In order to construct a timed cube approximation, it is necessary to determine the stability of internal signals using some form of timing analysis. There are essentially two related approaches that can be applied. The first approach is to use a method that finds the time separation between events such as those described in [17]–[20]. The second approach is to use a state-based approach in which timing information is represented using *difference bound matrices* (DBMs) or *zones* [2]–[7]. While time-separation-based methods have been used for analyzing both timed circuits and burst-mode circuits, they do have their drawbacks, i.e., they are substantially more complex when specifications include either choice or circuits with disjunctive behavior, such as OR gates. These methods can also be overly conservative at times when the time separation is state dependent. In our experience, we have found that time-separation methods are efficient when only a few time separations are required, but they are very inefficient when many are required such as when finding the reachable states of a timed specification. It is for these reasons that synthesis tools for timed circuits utilize zone-based representations during state-space exploration. The ATACS tool, to which we have added the timing verifier described in this paper, is one such zone-based tool [2], [3], [5]. Since the goal of this paper is to utilize this timing verifier during technology mapping, the zone-based representation of the state space is already available from the preceding synthesis step. For this reason, we decided that performing the timed stability analysis beginning from the zones associated with a state would be the most efficient approach. These zones already contain partial state-dependent time-separation information. Starting with any zone, a modified timed state-space exploration can determine the amount of time elapsed while following a sequence of states. Our method starts in a state in which an internal signal changes evaluation; when the amount of time elapsed is found to be larger than the worst-case delay of the logic, the internal node is known to have stabilized. Our experimental results show that for the circuits that this paper targets (namely, those produced by our technology mapper), the analysis is very efficient and produces very few false negative results.

II. Background Terminology

The verifier described in this paper takes as input a *time Petri net* (TPN) that defines the circuit and the behavior of the environment and a *netlist* that represents the circuit to be verified. The verification procedure also creates a *state graph* (SG) to represent reachable timed states. The goal of this verification procedure is to identify *hazards* in the circuit being verified. This section describes each of these terms formally.

A. TPNs

Our method uses TPNs [21] to model the possible input behaviors and the required output behaviors for timed circuits. Let W be a finite set of wires in a timed circuit. The 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 wire w. In the following definitions, let $\mathbb{Q}^+$ and $\mathbb{R}^+$ denote the sets of nonnegative rational and nonnegative 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:

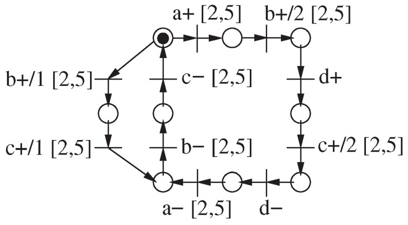

Fig. 1. TPN for our running example.

- $W = I \cup O$ 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;
- $l : T \to \mathbb{Q}^+$ is the lower timing bound function;
- $u : T \to \mathbb{Q}^+ \cup \{\infty\}$ is the upper timing bound function;
- $L : T \to W$ 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 $D : T \to \mathbb{R}^+$ is a clock assignment function that assigns nonnegative real-valued ages to transitions. With every transition $t \in T$, its associated *preset* is $\bullet t = \{p \in P | (p,t) \in F\}$. The *postset* of a transition is defined as $t\bullet = \{p \in P | (t,p) \in F\}$. 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 Fig. 1. 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 indicate different transitions on the same signal wire. If $a+$ fires, then $b+/2$ becomes enabled and fires within 2 to 5 more time units, enabling $d+$. Note that the timing on the firing of $d+$ is specified in the netlist, which is defined next.

The TPNs are restricted to one-safe nets to simplify the software, but this is not a fundamental restriction. Any TPN with a finite state space can be analyzed. This includes any arbitrary choice constructs in the TPN as well. The state space generated for the TPN though must be *output semimodular.* A state space is output semimodular when the only transitions that can be disabled without firing are inputs, and they are only disabled by other inputs. Our method cannot analyze the hazard behavior of gates such as arbiters since the hazard analysis searches for gate disablings, which are inherent in these gates.

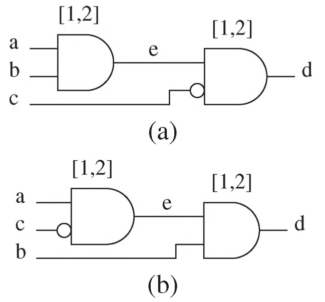

Fig. 2. (a) Netlist that is hazardous under the speed-independent model. (b) Netlist that is hazard-free under the speed-independent model.

B. Netlists

The goal of this paper 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.

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 that 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 implementation of signal d in our example is shown in Fig. 2(a). 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 $f_e(a, b) = \text{AND}(a, b)$, which has a delay of 1 to 2 time units. An alternative circuit implementation for signal d is shown in Fig. 2(b). While this model assigns delays to gates, delays on wires are easily supported. Consider a signal x that forks to two different gate inputs. Differing delays on these wires can be modeled by introducing a fictitious buffer on each branch of the fork and then assigning these delays to the buffers. These buffers are then included when determining the maximum delay path from primary inputs to outputs.

The verification method described in this paper requires that the primary outputs cut the circuit. In other words, if all primary outputs are removed from the netlist, the netlist would become acyclic. Intuitively, this means that there can be no internal cycles in the netlist. If there are internal cycles, then to apply this method, nodes from N would need to be moved into O such that the circuit would now be cut by its primary outputs. Most synthesis tools actually satisfy this requirement. Furthermore, since the goal of this paper 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 paper to the case where internal feedback is allowed.

C. SGs

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 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 $= \langle S, T_{\text{SG}}, \delta\rangle$, where:

- S is the set of states;
- T_{SG} is the set of transitions;
- $\delta \subseteq S \times T_{\text{SG}} \times S$ is the set of state transitions.

Each 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 is described using zones that are typically represented using DBMs [22]. 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 (denoted $\max(z, t_i, t_j)$). The minimal time separation after t_j before t_i fires (i.e., $\min(z, t_i, t_j)$) is equal to $-z_{ji}$. The notation $\text{var}(z)$ is used to denote the set of variables in zone z. An example zone for the point right after $a+$ fires, which represents the relationship $2 \leq \tau_{a+} - \tau_{c-} \leq 5$, is given by

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

Using a timed state-space exploration algorithm such as the ones in [3] and [5], 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 is derived. In other words, a SG generated from a CGE version abstracts the internal nodes to reduce the verification complexity. The CGE circuit for both netlists in Fig. 2(a) and (b) is shown in Fig. 3(a). The SG found using this circuit and the TPN in Fig. 1 is shown in Fig. 3(d). Using ν, each state vector is labeled in the SG to show the value of all signal wires. The zones calculated during the timed state-space exploration are omitted for clarity. Each edge of the SG is labeled with a signal transition $t \in T$. In other words, T_{SG} is equal to transition set T from the TPN. 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 detail to note is that during state-space exploration to derive this SG, this 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 Fig. 3(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.

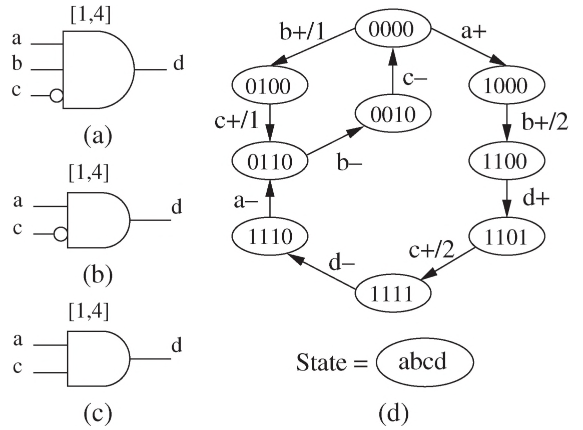

Fig. 3. (a) Correct CGE circuit. (b) Incorrect CGE circuit. (c) Another incorrect CGE circuit. (d) SG for correct CGE circuit.

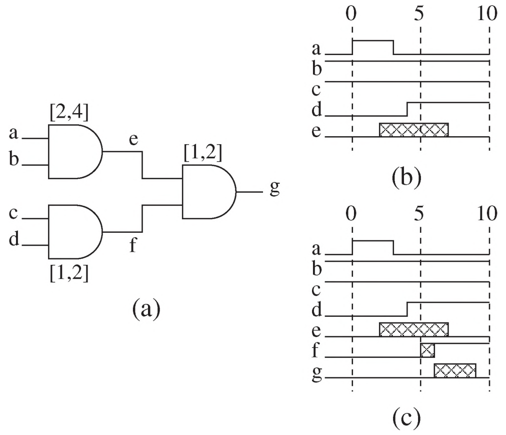

Fig. 4. (a) Example circuit to illustrate hazards. (b) Timing scenario for an acknowledgment hazard. (c) Timing scenario for a monotonicity hazard.

D. Hazards

Hazards can manifest in asynchronous circuits due to violations in the *acknowledgment* or *monotonicity* properties [10]. This subsection describes these hazard conditions briefly using a simple example. Both acknowledgment and monotonicity hazards are defined algorithmically in Section III-C.

An acknowledgment hazard occurs when an internal node becomes excited to change to a new value, but the conditions that caused the excitation change before the node can be shown to have become *stable*. A node is said to be stable when it equals its evaluation. In other words, it is not enabled to change. An example of a circuit with an acknowledgment hazard is shown in Fig. 4(a), and the hazard manifests under the timing shown in Fig. 4(b). This three-gate circuit implements the function $g = abcd$. The output g should never be enabled to go HIGH during the time period depicted in Fig. 4(b) since there is no time point in which all four inputs are simultaneously HIGH. At time zero, input signals a and d are LOW, while b and c are HIGH. This forces the internal nodes e and f and the output g to be LOW. At time zero, input a changes to HIGH and stays HIGH for 3 time units. This enables node e to rise. However, since the delay of the AND gate driving node e has a time delay between 2 and 4 time units, it is not certain whether or not node e actually rises before signal a goes LOW. This represents an uncertainty on node e in response to the pulse on a, and the possible failure to *acknowledge* the transition on a indicates an acknowledgment hazard on node e. If input timings and gate delays allow this possible glitch on e to propagate to the output, the primary output g may have a monotonicity hazard as described below.

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

Fig. 5. Top-level algorithm for verification.

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 nonmonotonically. 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 that is stable LOW cannot be found while the other input is rising, it is possible that the AND gate may momentarily evaluate to 1, causing a glitch on its output. In the example shown in Fig. 4, it is possible that the glitch on node e discussed previously can propagate to the output. Consider the timing diagram shown in Fig. 4(c). After input a goes LOW at time 3, input d rises at time 4. This causes node f to be enabled to rise and can do so as early as time five. At time five, both inputs to the AND gate driving the output are in transition, and there is the possibility for a glitch on the output. This timing scenario represents a monotonicity hazard on the output g.

III. Verification Algorithm

In [10] and [23], the following theorem giving sufficient conditions for correctness of a determinate 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 $\text{NET} = \langle V, E\rangle$ be a circuit implementing the behavior described by $\text{TPN} = \langle W, T, P, F, M_0, s_0, l, u, L\rangle$. The NET is a correct implementation of the TPN if 1) it is CGE to the TPN and 2) it satisfies the acknowledgment and monotonicity properties.

Our verification algorithm shown in Fig. 5 checks these correctness conditions. This algorithm takes as input a TPN to represent the possible input behavior and the required output behavior and a NET to represent the circuit to be checked. When the circuit is not correct, this algorithm reports the locations of the errors that it finds. This section describes this algorithm in detail.

A. Checking Equivalence

The `check_equivalence` function forms a CGE netlist, uses this netlist and the TPN to derive a SG, and checks if the CGE netlist provides outputs when and only when allowed by the TPN. 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 expressed 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, this 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 this example, the CGE representation for the netlists in Fig. 2(a) and (b) is shown in Fig. 3(a). This gate's delay is [1, 4] since in both cases, there exists a minimum delay path of one and a maximum delay path of four.

Using this CGE netlist and the given TPN, a SG is found using a timed state-space exploration algorithm. During this state-space exploration, output firings are checked. If an output fires prematurely, such as in the example shown in Fig. 3(b), an error is reported to the user. One interesting fact about this circuit though is if the minimum delay of this gate is increased to six, then this circuit would actually be correct as timing would guarantee that d always rises after $b+/2$ fires as required by the TPN. If during the analysis, an output is expected and the circuit does not provide one, an error is also reported. In our example, if the function shown in Fig. 3(c) is used, after $a+$ and $b+$ fire, a $d+$ is expected, but the circuit does not produce it. This check models a progress condition similar to *completeness with respect to specification* [9]; and *strong conformance* [12]. When no errors are detected, `check_equivalence` returns a SG.

B. Finding Stable States

After the `check_equivalence` step, this 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 the internal signal behavior implicitly. In particular, the `find_stable_states` algorithm determines 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 $(s, t, s') \in \delta$. The predicate `stable`(s, n) is defined to return TRUE when we have been able to determine through our analysis that node n is stable HIGH or LOW in state s. It returns FALSE when either node n is found to be changing in state s or the algorithms are unable to determine that the node is certainly stable. In other words, it is a conservative approximation of the stability of each node in each state. Similarly, the predicate `stable`(s, n, s') is defined as a conservative approximation of the stability of node n during the state transition from state s to state s'. These stability predicates are used, as shown in Section III-C, to determine if there are any hazards in the given netlist.

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

Fig. 6. Algorithm for finding stable states.

```
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
```

Fig. 7. Untimed stabilization algorithm.

The algorithm to find the stability information is shown in Fig. 6. 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 is 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 Fig. 3(d) and netlist in Fig. 2(a), `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.

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 the stability of the internal signals, whenever possible. In the next subsection, a brief review of untimed stabilization, which comes from the work in [10], is given, followed by a detailed discussion of 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.

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 Fig. 7. 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 transition t could only have occurred if the internal node n is stable at its Boolean evaluation, then it can be said that transition t has acknowledged that node n is stable.

To determine if an internal node n is acknowledged to be stable by a state transition (s, t, s'), the `exists_path` function is used first to check if a path exists from n to the output transition under consideration. The `must_prop` function is then used to check 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 noncontrolling stable values on the side inputs. Consider the example netlist in Fig. 2(a) and the state transition (1100, $d+$,1101). There exists a path between node e and 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, $\texttt{stable}(1100, 1101, e)$ is set to TRUE.

The `distribute` function is used to copy this stabilization forward in the SG until a change in evaluation is encountered. In particular, $\texttt{stable}(1100, 1101, e)$ implies that the following stability conditions are TRUE: $\texttt{stable}(1101, e)$, $\texttt{stable}(1101, 1111, e)$, $\texttt{stable}(1111, e)$, $\texttt{stable}(1111, 1110, e)$, $\texttt{stable}(1110, e)$, and $\texttt{stable}(1110, 0110, e)$. 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 complexity of the `distribute` function is $O(|S|)$, where $|S|$ is the number of states in the SG.

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 d. As explained later in the text, this lack of stabilization on the falling transition of e indicates a hazard on node e.

A similar analysis of the circuit in Fig. 2(b) finds the rising transition on node e acknowledged by $d+$ and the falling transition acknowledged by $d-$ since b is HIGH (a noncontrolling value) when d goes LOW. As a result, this circuit is hazard-free under the speed-independent model.

2) Timed Stabilization: When timing information is taken into account, the hazard found for the netlist shown in Fig. 2(a) may not actually manifest. If this is the case, then node e is hazard-free. This section describes our new method to determine stabilization using timing information. Timed stabilization attempts to show further stability in the SG 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 SG. When it can be shown that in the worst case, a sufficient amount of time has elapsed, node n can be stabilized.

```
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)
```

Fig. 8. Timed stabilization algorithm.

```
update_zone ( $TPN, NET, z, t_i$ )
  $z'$ = extend($z, t_i$)
  found_causal = FALSE
  foreach $t_j \in var(z')$ 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($z'$)
```

Fig. 9. Algorithm to update the zone.

The algorithm to determine timed stabilization is shown in Fig. 8. For each node n, the algorithm first measures the longest path delay from any primary input to node n and also from any primary output, which is an input to the cone of logic that produces the signal 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 Fig. 2(a), this delay for e is determined to be 2. This is accomplished using the `find_max_delay` function, which computes the longest path in a directed acyclic graph. Its complexity is $O(|V + E|)$. 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 the 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 SG where the node n becomes unstable. Note that t_i is the transition on a primary input or output signal that feeds the cone of logic that ends with the node n, and the change in this signal's value is what causes n to become unstable. The algorithm then takes the zone z associated with state s and updates it to include transition t_i.[1] 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 SG. Finally, the algorithm initializes a `path` array, which is used to terminate cycles during the analysis of a path in the SG.

The `update_zone` algorithm shown in Fig. 9 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, for this extended zone z', it searches $\mathrm{var}(z')$, 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 t_j that it finds is the causal transition for t_i. The maximal time separation of t_i from t_j, z'_{ij} is either the upper timing bound taken from the TPN when t_i is a transition on an input wire or the maximum delay in the netlist generating t_i when it is a transition on an output wire. For all transitions t_j that enable t_i, the minimal time separation of t_i from t_j, $-z'_{ji}$, is either the

[1] We are using a past variable style zone algorithm, so transitions are only added to the zone after they fire.

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

(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)

Fig. 10. Zone creation and evolution.

```
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
```

Fig. 11. Timed stabilization recursion.

lower timing bound taken from the TPN or the minimum delay in the netlist. The minimum delay in the netlist is computed using the `find_min_delay` function, which finds the shortest path in a directed acyclic graph and has complexity $O(|V + E|)$. For the other transitions t_j, z'_{ij} and z'_{ji} are set to infinity. 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 can determine these timing relationships. As an example, the zone found for the state 1110 in our example is shown in Fig. 10(a). The new zone after adding transition $a-$ is shown in Fig. 10(b).

The `do_timed` algorithm shown in Fig. 11 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 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 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 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 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 that 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 state s.

Our algorithm has the potential for requiring the exploration of a large number of paths. In the worst case, $O(|S|!)$ paths would need to be analyzed. This complexity bound, however, requires the SG to be fully connected and all states to be explored before a timed stabilization is encountered. In reality, SGs are typically very sparsely connected, and the time accumulated during the traversal would accumulate beyond the maximum delay of the circuit being analyzed well before all states are considered. Therefore, assuming that the average number of successors for a state is x and the average depth required by analysis is y, then the number of paths that would typically need to be explored is $O(x^y)$. While this can still be a very substantial number, in the experimental results reported in Section V, the average value for x is 1.29, and that for y is 3.41. This means that each call to `do_timed` tends to only need to explore two or three paths. If the other examples require the algorithm to recurse further through the SG before finding a point of timed stabilization, then there is the potential that a substantial number of side paths would need to be explored. For such examples, the value of y can be fixed, meaning that the algorithm would be limited in the path length that would be explored. This can improve efficiency at the potential cost of more false negative results.

It is now useful to again consider the example netlist in Fig. 2(a). 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 Fig. 10(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 Fig. 10(c). The timing of the other nondiagonal entries is set to ∞. The zone is then recanonicalized, and the resulting zone is shown in Fig. 10(d). The parameter of interest is the minimum elapsed time between the last transition entered, $b-$, and the initial transition $a-$, which is two in this case. Note that lower bounds appear as negative values in a DBM. Since 2 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 Fig. 10(e), and recanonicalizes to obtain the zone

```
check_acknowledgment (SG, NET)
  foreach (n ∈ N)
    foreach (s, t, s') ∈ δ
      if ((eval(s,n) ≠ eval(s',n)) and
      (not stable(s,s',n))) then
      report acknowledgment hazard on
      node n for (s,s',n)
```

Fig. 12. Algorithm to check for acknowledgment hazards.

```
check_monotonicity (SG, NET)
  foreach n ∈ (N ∪ O)
    foreach s ∈ S
      if (f_n(cube(s)) = X) then
        foreach v ∈ FI(n)
          if (potential_hazard(s,n,v)) then
            report monotonicity hazard on
            node n for (s,v)
```

Fig. 13. Algorithm to check for monotonicity hazards.

shown in Fig. 10(f). The new minimum time elapsed from $a-$ to $c-$ 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, as shown in the next section.

C. Checking for Hazard-freedom

As described earlier, hazards can manifest in asynchronous circuits due to violations in the acknowledgment or monotonicity properties [10]. This section explains how our method checks for violations of these two properties. While the theory used is essentially the same as that in [10], this section introduces new algorithms to perform these checks.

The algorithm shown in Fig. 12 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 has not stabilized before it changes evaluation, then an acknowledgment hazard is reported. For the netlist shown in Fig. 2(a), using only untimed stabilization, a hazard is found on node e for the state transition (1000, $b+/2$, 1100). Timed stabilization, however, detects that this state transition is stable for node e, so it is hazard-free when timing is considered.

The algorithm to check a netlist for monotonicity violations is shown in Fig. 13. Monotonicity violations are caused on a node $n \in (N \cup O)$ by one of its fanins $v \in FI(n)$ in a particular state $s \in S$. For each node, all states in the SG are applied to the netlist. At this point, the algorithm constructs a cube formed from all $n \in (I \cup O \cup N)$ and applies it to f_n to determine if, given what is known about the current state, the value of node n is being forced to a known value. The function $\texttt{cube}(s)$ is a cube formed from the values determined on each $n \in (I \cup O \cup N)$ using the state vector and what is known about internal nodes from the `stable` and `eval` predicates.

```
potential_hazard (s, n, v)
  if (v ∈ N ∧ stable(s,v)) then
    return FALSE
  if ((v ∈ (I ∪ O)) and
    (eval(s,n) ≠ eval(bitcomp(s,v),n))) then
    return FALSE
  if ((!stable(s,n)) and
    (f_n(pcube(s,v)) = 'X')) then
    return FALSE
  if (f_n(pcube(s,v)) = eval(s,n)) then
    return FALSE
  return TRUE
```

Fig. 14. Algorithm to check for a potential hazard.

Note that the value returned by $\texttt{cube}(s)$ represents a set of implementation states. More formally, $\texttt{cube}(s)$ is defined for each node v as

$$\texttt{cube}(s)(v) = \begin{cases} s(v), & \text{if } v \in I \cup O \\ \texttt{eval}(s, v), & \text{if } v \in N \wedge \texttt{stable}(s, v) \\ X, & \text{otherwise} \end{cases}$$

where $s(v)$ denotes the value of signal v in state s. Note that "X" means that the value of the node is unknown.

Next, $\texttt{cube}(s)$ is applied to the function $f_n(v_1, v_2, \ldots v_r)$. $f_n(\texttt{cube}(s))$ is written to denote $f_n(\texttt{cube}(s)(v_1), \texttt{cube}(s)(v_2), \ldots, \texttt{cube}(s)(v_r))$. In other words, the value of each fanin in the cube is extracted and applied to the function f_n. Since some values applied to the function may be unknown, the function f_n may return X. For example, if $f_n(a, b) = \text{AND}(a, b)$, $f_n(0, X) = 0$, while $f_n(1, X) = X$.

If the value at node n is determined to be unknown, then all fanins of node n are checked for potential hazards. The algorithmic definition for `potential_hazard` shown in Fig. 14 is modified from [10] to fit the definitions used in this paper. To help in the evaluation of potential hazards, a potential cube $\texttt{pcube}(s, v)(u)$ is formed as follows:

$$\texttt{pcube}(s, v)(u) = \begin{cases} \texttt{cube}(s)(v), & \text{if } u \neq v \\ \texttt{eval}(s, u), & \text{if } u = v \end{cases}.$$

The potential cube for s and v is equivalent to $\texttt{cube}(s)$, except at node v, which is set to its final evaluation. The `bitcomp` function referenced in Fig. 14 takes as arguments a state s and node $v \in (I \cup O)$ and returns a new state that has the bit v complemented. This new state and node n then become arguments to the predicate `eval`.

The absence of a potential hazard is determined by examining the conditions that prevent it from occurring. There are four such conditions shown in the algorithm of Fig. 14 and briefly described here. A potential hazard cannot occur on node n in state s for fanin v:

1) if v is an internal node and is stable in state s;
2) if v is a primary input or output and the evaluation of n changes when v is complemented;
3) if n is not stable in state s and $f_n(\texttt{pcube}(s, v))$ does not indicate that n is being forced to a known value;
4) if $f_n(\texttt{pcube}(s, v))$ is equal to the Boolean evaluation of n in state s.

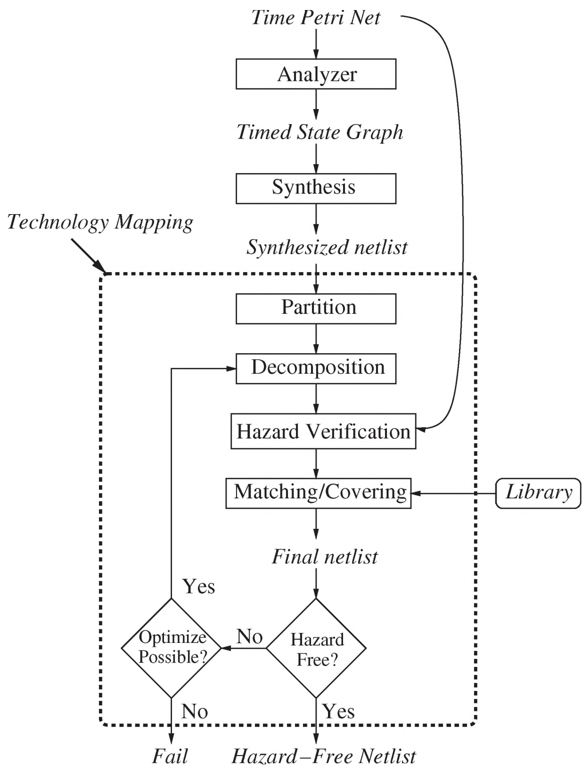

Fig. 15. Technology mapping design flow.

Condition 1 implies that potential hazards can only be caused by internal nodes if they are unstable. Condition 2 implies that potential hazards are only caused by external nodes when a change in their values does not result in a change in evaluation. Condition 3 indicates that there is no potential hazard when node n is not being forced to some value if n is not stable to begin with. Finally, condition 4 implies that there is no potential hazard if setting node v to its final evaluation forces node n to its final evaluation. If all four of these conditions cannot be met, then a potential hazard exists, and a monotonicity hazard is reported on node n in state s caused by fanin v.

IV. Technology Mapping

We have incorporated the gate-level hazard verifier presented in this paper within a hazard-aware technology mapper for timed circuits. This section gives an overview of our technology mapper, but more detail can be found in [24]. *Technology mapping*, which is also called *library binding*, is the process whereby a *technology-independent* logic representation is mapped to a *technology-dependent* library. As shown in Fig. 15, our technology mapper takes as inputs a TPN, a synthesized netlist, and a gate library and, in most cases, outputs a hazard-free netlist composed of library elements. The technology mapping process combines the steps of partitioning, decomposition, hazard verification, and matching/covering. If the technology mapper is unable to find a hazard-free solution, there are a number of optimizations that can be applied [24], and the process is repeated until a hazard-free implementation is found; otherwise, a failure is reported to the user.

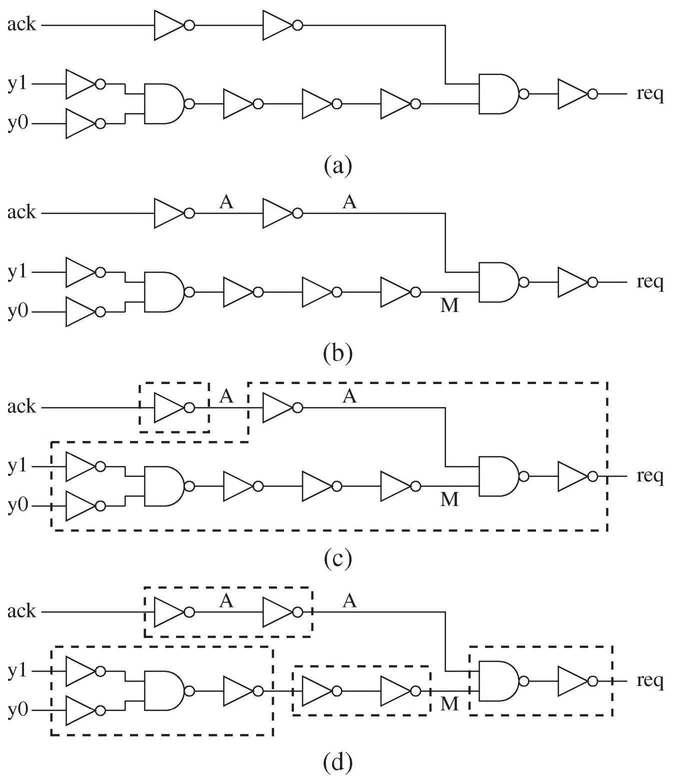

Fig. 16. Technology mapping example. (a) Example circuit *req*. (b) Circuit annotated with hazards. (c) Cover based on area minimization. (d) Cover based on hazard awareness.

Partitioning of the synthesized netlist is unnecessary within the framework of our research because the form of the synthesized netlist is a single-output cone of logic. Decomposition transforms the synthesized netlist into a logically equivalent network called the *subject graph*, which consists entirely of *base functions*. Typical base functions include inverters, two-input NAND gates, and a state-holding device. For asynchronous applications, the state-holding device is a Muller C-element (CEL). We employ the same decomposition process used in synchronous systems, i.e., repeatedly applying DeMorgan's laws and the associative law to the subject graph.

Each cell in the gate library is modeled as an atomic gate. The combinational and sequential cells are single-output logic functions. The sequential elements are either CELs or generalized CELs (gCs) with internal feedback for state-holding purposes. A gC incorporates both logic and state holding in one atomic gate. The same decomposition algorithm that is applied to the synthesized netlist is also applied to each cell in the library, resulting in a *pattern graph* for each library cell. This insures that a subgraph of the subject graph can then be structurally matched to the pattern graphs in the library. Inverter pairs are often inserted in the decomposed netlist [25] to increase granularity, which provides more options in the matching/covering stage.

An example subject graph is shown in Fig. 16(a). This synthesized netlist is the output *req* from the benchmark *alloc-outbound*. The logic equation describing this netlist is $req = ack \cdot \overline{y1} \cdot \overline{y0}$. The subject graph in Fig. 16(a) is shown after decomposition and with inverter pairs inserted. The subject

TABLE I
COMPARISON OF STANDARD BENCHMARKS AGAINST OTHER TIMING VERIFICATION TOOLS

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

graph is then verified using the gate-level algorithms presented in this paper. Fig. 16(b) shows how the nodes are annotated with the type of hazards determined during verification. An A means that an acknowledgment hazard is found on that node, and an M indicates that that node causes a monotonicity error at the output of the gate to which it is a fanin.

The final step of technology mapping is to match pattern graphs to the subgraphs of the subject graph. This matching process is typically optimized for a parameter such as area or delay. For instance, Fig. 16(c) shows how our example would be matched when optimized for area. Here, it is assumed that each element in the library is assigned an area and delay number. This singular focus on area ignores the consequences of covering circuits with hazardous nodes. The area covering leaves one acknowledgment hazard exposed, and verification performed on the resulting netlist indicates that this acknowledgment hazard is still present in the final netlist. In addition, a new monotonicity hazard has been created on the output *req* due to timing relationships that have changed in the final netlist. This circuit therefore is hazardous.

Fig. 16(d) shows how the same netlist would be covered if hazard-freedom is the primary cost factor. Our experience has shown that if the hazard-aware technology mapper attempts to encapsulate acknowledgment hazards and leave nodes with monotonicity hazards exposed in the covered netlist, then the resulting circuit is usually hazard-free. For the example in Fig. 16(d), hazard-aware covering encapsulates one acknowledgment hazard and leaves the node with the monotonicity hazard exposed. The node with the exposed acknowledgment hazard is not a problem because the inverter pair preceding this node is removed from the final netlist (since its implementation is just a wire), and the node with the acknowledgment hazard becomes the input node *ack*. By definition, primary inputs cannot have hazards. Therefore, verification on this newly covered netlist is found to be hazard-free.

V. EXPERIMENTAL RESULTS

The gate-level timing verification method described in this paper has been tested on numerous examples. Table I compares our new gate-level timing verification method using standard benchmarks against results for the timed automata tool KRONOS [6], a conservative approximation method described in [4], and the ATACS explicit-state timing verifier [26]. All runtimes are specified in seconds. For KRONOS runtimes, an entry with a greater than sign (>) indicates the amount of time after which the verification ran out of memory. The runtimes for KRONOS and Peña *et al.*'s methods are taken from their papers, while the runtimes for ATACS and our new method are from a 900-MHz Pentium 4 with 256 MB of memory.[2] For our new method, 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, 25 times faster than Peña *et al.*'s tool, and twice as fast as the explicit state method used 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 runtime 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 state method in ATACS is configured to continue after finding one hazard and identify all hazards. The number of hazards found is reported in the hazards column under ATACS. Since the explicit state method is exact, the hazards column represents the number of true hazards in these

[2] We selected this computer as it has a processor with comparable performance to the ones available when KRONOS and Peña *et al.*'s results were generated.

TABLE II
COMPARISON FOR DECOMPOSED NETLISTS

		ATACS			New Method		
Example	Gates	Time(s)	Mem(MB)	Hazards	Time(s)	Mem(MB)	Hazards
scsiSV	18	1.35	7.9	0	0.13	1.3	0
slatch	29	33.5	53.4	0	0.15	1.8	0
lapbsv	37	20.0	41.5	0	0.17	1.3	0
elatch	38	183	229	0	0.28	1.8	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

circuits. It should be noted that KRONOS does not check for hazards but instead is only checking conformance, while Peña *et al.*'s tool halts after a hazard is found. Since all these methods are exact, if configured to return all hazards found, they would all produce the same number of hazards. The number of hazards found by the new method is also reported in the last column. When the number of hazards in our new method is larger, this indicates that our method found additional false hazards. Despite being a conservative approximation, our method found the exact number of hazards in most cases. However, in the three examples *rpdft*, *sbuf-ram-write*, and *sbuf-send-pkt2*, our new method found one additional false hazard. It should also be noted that six benchmark circuits included internal cycles, which meant that our method cannot be applied. The exact methods do not have this limitation.

The key advantage of our new method is its ability 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 are selected, and gate-level circuits that use only two-input NAND gates, inverters, and CELs are derived for them. This is accomplished using the decomposition procedure within our technology mapper described in Section IV. Our results are shown in Table II. In all the examples, our method is still able to check for hazards in 1.2 s or less, while for the largest examples, the explicit state method cannot complete.

Our new verification method uses a combination of timed and untimed algorithms to determine hazard-freedom for each node and each output in a netlist. It is found that stabilizations in the SG due to timing occur much more frequently than do stabilizations using untimed (speed-independent) methods. These results are shown in Table III. They are not surprising because the delays used for the basic circuit elements are fairly small (but physically practical), so circuit delays in the decomposition, up to the node of interest, are often small, and stabilization through the SG occurs reasonably quickly.

The surprising result from Table III is how effective the timed stabilization algorithms are. The numbers in the last three columns indicate how many nodes in each circuit are found to be hazardous. When untimed stabilization alone is used, in all cases but one, over half the nodes are hazardous. When only timed stabilization is used, this number is reduced considerably. The last column indicates the results achieved by first running timed stabilization, followed by untimed stabilization. Note how timed stabilization alone gives identical results to the case where timed stabilization is followed by untimed stabilization. This is a potentially significant finding in that it says, at least for these examples, that there is no need to do untimed stabilization. Since untimed stabilization may need to be iterated (unlike timed stabilization), a cost savings in computation time, without apparent loss of accuracy, occurs if timed stabilization is run by itself.

TABLE III
HAZARDS FOUND BASED ON THE STABILIZATION METHOD

		One method		Timed/
Example	Gates	Untimed	Timed	Untimed
alloc-outbound	11	6	0	0
chu133	9	7	1	1
converta	12	8	2	2
ebergen	9	4	3	3
half	7	7	1	1
mp-forward-pkt	10	6	0	0
nowick	10	7	0	0
ram-read-sbuf	17	13	0	0
rcv-setup	6	5	0	0
rpdft	8	8	2	2
sbuf-ram-write	17	12	2	2
sbuf-read-ctl	10	6	0	0
sbuf-send-ctl	13	11	1	1
sbuf-send-pkt2	13	11	1	1
trimos-send	24	18	5	5

VI. HAZARD ISSUES

There are two issues that arise, warranting further discussion: false hazards and nonpropagating internal hazards.

A. False Hazards

In a small number of cases, our conservative method reports that a node is hazardous when a full-timed state-space exploration indicates that there is no hazard present. These *false hazards* are a result of the abstraction method, which limits the visible states to those contained in the CGE SG. Between any two states in this SG, a number of internal signals can be undergoing an ordered sequence of transitions. The stabilization algorithms do not always find internal nodes to be stable, and if more than one input to a gate is unstable in the same state and a forcing side input cannot be found, then a monotonicity hazard is reported.

An example circuit to illustrate a false hazard is shown in Fig. 17. This circuit is from the *rpdft* example in the suite of examples used by the timed automata tool KRONOS [6]. Here,

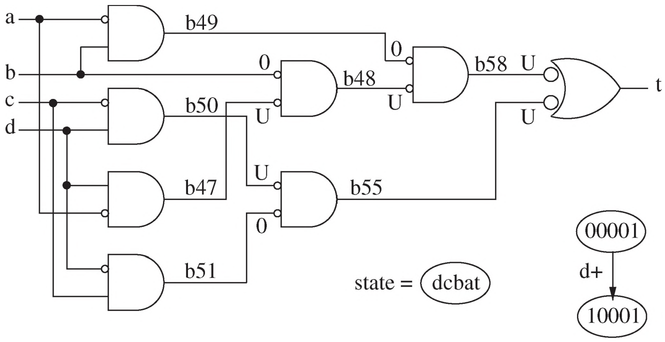

Fig. 17. False hazard example using circuit *rpdft*. Assume that gates have a delay of [1, 1].

a false monotonicity hazard is reported on output node t in state 10001 caused by fanin $b58$. This is seen by starting in state 00001, where the external signals are stable at the values in the state vector. These stable values force signal $b58$ to be LOW and signal $b55$ and output t to be HIGH. When signal d rises and the circuit moves to state 10001, signal $b55$ is enabled to fall (through two gate delays) and signal $b58$ is enabled to rise (through three gate delays). However, the algorithm is not able to determine the order in which these internal nodes actually switch. Thus, in state 10001, neither $b55$ or $b58$ has stabilized, and a monotonicity hazard on output t is reported.

After the timed and untimed stabilization has been completed for the circuit in Fig. 17, the stability information for state 10001 is shown on each internal node. For the internal signals, a U indicates that this node is unstable, and a 0 indicates that the node has stabilized at that value. Note also that all gates have at least one stable input, except for the gate driving the output t.

To explore why this hazard is false, the full-timed SG for the region of interest must be examined. This SG is shown in Fig. 18. Note that between the time signal d rises when state 10001 is entered and signal b rises when state 10001 is exited, internal signals $b47$, $b50$, and $b58$ rise, and signals $b55$ and $b48$ fall. It is clear from this SG that signal $b55$ falls before $b58$ rises. Thus, there is no actual state in this SG where the ambiguity in Fig. 17 is present.

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 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.

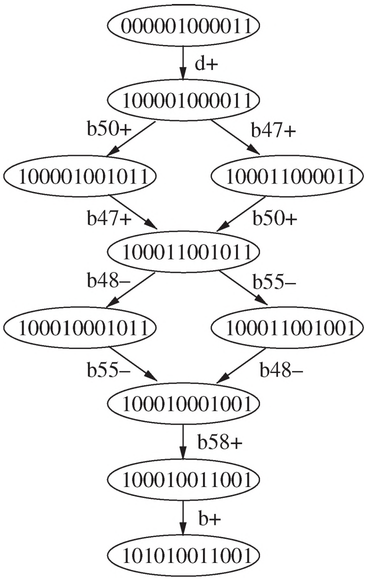

Fig. 18. Full-timed SG for the region of interest in circuit *rpdft*. The state vector is $\langle d, c, b, a, b47, b48, b58, b50, b51, b55, t \rangle$.

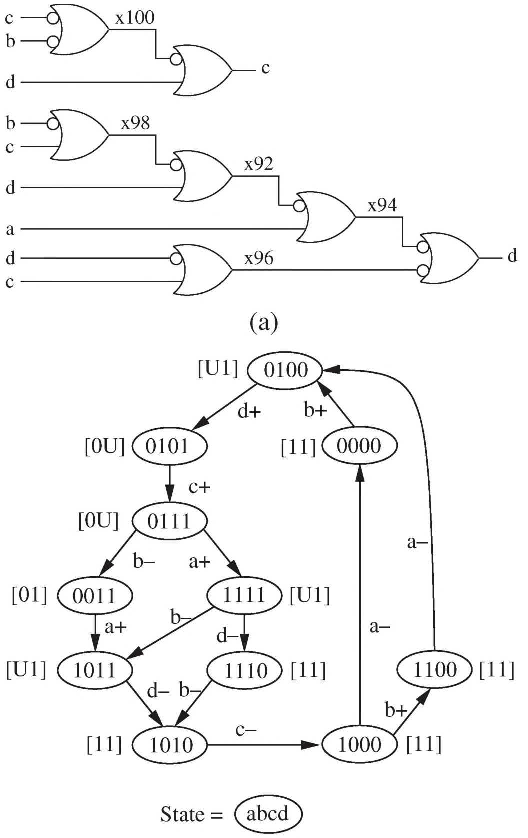

Fig. 19. Nonpropagating acknowledgment hazard example. (a) *half* circuit. (b) *half* state graph. Assume that all gates have a delay of [1, 1].

B. Non-propagating Internal Hazards

The intent of the verification portion of this paper is to identify nodes where hazardous behavior is occurring. However, it is known that hazardous activity on internal nodes does not necessarily mean that the circuit fails. In other words, if hazards on internal nodes do not propagate to the output, the circuit as a whole may not be hazardous.

An example of one such circuit (there are many) is shown in Fig. 19(a). This circuit is called *half* and is taken from the examples used by the KRONOS tool. Fig. 19(b) shows the SG for the *half* circuit. Note that in the SG, stability information for internal nodes $x94$ and $x96$ is placed in square brackets next to each state. After verification, it is found that node $x96$ has an acknowledgment hazard between states 0101 and 0111. This is seen in the SG by noticing that node $x96$ is unstable in both of these states, but its evaluation changes, i.e., $x96$ evaluates to 0 in state 0101 but evaluates to 1 in state 0111. As shown in the algorithm of Fig. 12, an acknowledgment hazard is reported between two states under these conditions.

After circuit verification, it is also found that output d is hazard-free. In other words, the acknowledgment hazard on node $x96$ did not propagate to the output. This is because the output is held in a HIGH state by node $x94$, which is stable at 0 during the state transition from 0101 to 0111.

The point of this example is that it may be possible to declare some circuits hazard-free even when there is hazardous activity on the internal nodes. One reason this is often the case is because of blocking side inputs such as in the example of Fig. 19. This topic is of keen interest because it may be possible to develop algorithms that identify nonpropagating hazardous activity that has no effect on the primary outputs.

VII. Conclusion

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 that is refined with a new timed stabilization procedure. This allows our method to avoid generating an explicit SG 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.

We utilized this hazard analyzer within a technology mapper for timed circuits [24]. In asynchronous circuits, hazards must be avoided, and care must be taken during technology mapping so as not to 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.

Acknowledgment

The authors would like to thank K. Stevens of the University of Utah for his feedback on this work.

References

[1] 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 J. Solid-State Circuits*, vol. 36, no. 2, pp. 217–228, Feb. 2001.

[2] 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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 6, pp. 769–786, Jun. 1999.

[3] W. Belluomini and C. J. Myers, "Timed state space exploration using POSETs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 5, pp. 501–520, May 2000.

[4] M. A. Peña, J. Cortadella, A. Kondratyev, and E. Pastor, "Formal verification of safety properties in timed circuits," in *Proc. Int. Symp. Adv. Res. Asynchronous Circuits and Syst.*, Apr. 2000, pp. 2–11.

[5] E. Mercer, C. Myers, and T. Yoneda, "Improved POSET timing analysis in timed Petri nets," in *Proc. 10th Workshop SASIMI*, Oct. 2001, pp. 127–134.

[6] O. M. M. Bozga, H. Jianmin, and S. Yovine, "Verification of asynchronous circuits using timed automata," in *Electronic Notes in Theoretical Computer Science*, vol. 65, O. M. E. Asarin and S. Yovine, Eds. Amsterdam, The Netherlands: Elsevier, 2002.

[7] T. Yoneda and H. Ryu, "Timed trace theoretic verification using partial order reduction," in *Proc. 5th Int. Symp. Adv. Res. Asynchronous Circuits and Syst.*, 1999, pp. 108–121.

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

[9] J. Ebergen and S. Gingras, "A verifier for network decompositions of command-based specifications," in *Proc. Hawaii Int. Conf. Syst. Sci.*, Jan. 1993, vol. I, pp. 310–318.

[10] P. A. Beerel, T. H.-Y. Meng, and J. Burch, "Efficient verification of determinate speed-independent circuits," in *Proc. ICCAD*, Nov. 1993, pp. 261–267. [Online]. Available: http://jungfrau.usc.edu/pub/iccad93.ps

[11] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky, "Analysis and identification of speed-independent circuits on an event model," *Form. Methods Syst. Des.*, vol. 4, no. 1, pp. 33–75, Jan. 1994.

[12] G. Gopalakrishnan, E. Brunvand, N. Michell, and S. Nowick, "A correctness criterion for asynchronous circuit validation and optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 11, pp. 1309–1318, Nov. 1994.

[13] O. Roig, J. Cortadella, and E. Pastor, "Hierarchical gate-level verification of speed-independent circuits," in *Asynchronous Design Methodologies*. Los Alaminos, CA: IEEE Comput. Soc. Press, May 1995, pp. 129–137. [Online]. Available: ftp://gaudi.ac.upc.es/pub/reports/DAC/1995/UPCDAC- 95-01.ps.Z

[14] ——, "Verification of asynchronous circuits by BDD-based model checking of Petri nets," in *Proc. 16th Int. Conf. Appl. and Theory Petri Nets*, Jun. 1995, vol. 815, pp. 374–391.

[15] H. Zheng, C. J. Myers, D. Walter, S. Little, and T. Yoneda, "Verification of timed circuits with failure directed abstractions," in *Proc. ICCD*, Oct. 2003, p. 28.

[16] H. Zheng, E. Mercer, and C. Myers, "Modular verification of timed circuits using automatic abstraction," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 22, no. 9, pp. 1138–1153, Sep. 2003.

[17] C. J. Myers and T. H.-Y. Meng, "Synthesis of timed asynchronous circuits," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 1, no. 2, pp. 106–119, Jun. 1993.

[18] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello, "An algorithm for exact bounds on the time separation of events in concurrent systems," *IEEE Trans. Comput.*, vol. 44, no. 11, pp. 1306–1317, Nov. 1995.

[19] S. Chakraborty, D. L. Dill, and K. Y. Yun, "Min-max timing analysis and an application to asynchronous circuits," *Proc. IEEE*, vol. 87, no. 2, pp. 332–346, Feb. 1999.

[20] ——, "Timing analysis of asynchronous systems using time separation of events," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 8, pp. 1061–1076, Aug. 1999.

[21] P. Merlin and D. J. Faber, "Recoverability of communication protocols," *IEEE Trans. Commun.*, vol. COM-24, no. 9, pp. 1036–1043, Sep. 1976.

[22] D. L. Dill, "Timing assumptions and verification of finite-state concurrent systems," in *Proc. Workshop Autom. Verification Methods for Finite-State Syst.*, 1989, pp. 197–212.

[23] P. A. Beerel, J. R. Burch, and T. H.-Y. Meng, "Checking combinational equivalence of speed-independent circuits," *Form. Methods Syst. Des.*, vol. 13, no. 1, pp. 37–85, May 1998.

[24] C. A. Nelson, "Technology mapping of timed asynchronous circuits," Ph.D. dissertation, Univ. Utah, Salt Lake City, UT, 2004.

[25] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology mapping in MIS," in *Proc. ICCAD*, 1987, pp. 116–119.

[26] 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. Asia and South Pacific Des. Autom. Conf.*, Feb. 2001, pp. 335–340.

Curtis A. Nelson (M'89) received the B.S. degree in engineering with a concentration in electrical engineering from Walla Walla College, College Place, WA, in 1978, the M.S.E.E. degree from Washington State University, Pullman, in 1986, and the Ph.D. degree from the University of Utah, Salt Lake City, UT, in 2004.

He is currently a Professor of engineering with the School of Engineering, Walla Walla College. His primary teaching responsibilities are in the areas of VLSI design, embedded systems, and computer architecture. His current research interests include verification of asynchronous circuits and power driven logic design.

Dr. Nelson was an SRC Fellow from 2001 to 2004.

Chris J. Myers (S'91–M'96–SM'04) received the B.S. degree in electrical engineering and Chinese history from the California Institute of Technology, Pasadena, CA, in 1991, and the M.S.E.E. and Ph.D. degrees from Stanford University, Stanford, CA, in 1993 and 1995, respectively.

He is currently a Professor with the Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT. He is the author of more than 70 technical papers and the textbook *Asynchronous Circuit Design* (Wiley, 2001). He is also a co-inventor on four patents. His research interests include algorithms for the analysis of real-time concurrent systems, analog error control decoders, formal verification, asynchronous circuit design, and modeling and analysis of genetic regulatory circuits.

Dr. Myers received the National Science Foundation (NSF) Fellowship in 1991, the NSF CAREER Award in 1996, and the Best Paper Award at Async'99.

Tomohiro Yoneda (M'85) received the B.E., M.E., and Dr.Eng. degrees in computer science from Tokyo Institute of Technology, Tokyo, Japan, in 1980, 1982, and 1985, respectively.

In 1985, he joined the staff of Tokyo Institute of Technology. In 2002, he joined the National Institute of Informatics, Tokyo, where he is currently a Professor. He is also a Professor with Sokendai, The Graduate University for Advanced Studies, Hayama, Japan, and a Visiting Professor at Tokyo Institute of Technology. From 1990 to 1991, he was a Visiting Researcher at Carnegie Mellon University, Pittsburgh, PA. His current research interests include formal verification of hardware and synthesis of asynchronous circuits.

Dr. Yoneda is a member of the Institute of Electronics, Information, and Communication Engineers of Japan and the Information Processing Society of Japan.