Verification of Timed Circuits With Failure-Directed Abstractions

Hao Zheng, *Member, IEEE*, Chris J. Myers, *Senior Member, IEEE*, David Walter, *Student Member, IEEE*, Scott Little, and Tomohiro Yoneda, *Member, IEEE*

***Abstract*—This paper presents a method to address state explosion in timed-circuit verification by using abstraction directed by the failure model. This method allows us to decompose the verification problem into a set of subproblems, each of which proves that a specific failure condition does not occur. To each subproblem, abstraction is applied using safe transformations to reduce the complexity of verification. The abstraction preserves all essential behaviors conservatively for the specific failure model in the concrete description. Therefore, no violations of the given failure model are missed when only the abstract description is analyzed. An algorithm is also shown to examine the abstract error trace to either find a concrete error trace or report that it is a false negative. This paper presents results using the proposed failure-directed abstractions as applied to several large timed-circuit designs.**

***Index Terms*—Abstraction, formal verification, timed circuits.**

I. Introduction

TIMED circuits are defined to be any circuit that is aggressively optimized using timing assumptions such that their correctness is dependent on these assumptions. Utilizing timing assumptions can produce circuits with a significant improvement in speed as demonstrated by their use in a gigahertz research microprocessor [gigahertz unit test site (guTS)] at International Business Machines (IBM) [1] and by the Revolving Asynchronous Pentium Processor Instruction Decoder (RAPPID) instruction-length decoder designed at Intel [2]. The correctness of these new timed-circuit styles is highly dependent upon their timing assumptions. Therefore, extensive timing verification is necessary during the design process.

State explosion is a serious challenge for state-space-exploration-based verification approaches. Many methods exist to address the state explosion problem. Symbolic model checking, as described in [3], represents the state space implicitly using binary decision diagrams (BDDs), and is able to handle systems with substantially increased sizes. Applying decision diagrams to timing verification has also been successful [4]–[6]. Since interleaving among the concurrent events is the main source of state explosion, a number of techniques have been proposed to reduce the number of interleavings to be explored using partial orders [7], [8]. There has also been some success in adapting these methods to timing verification [9], [10]. While both decision diagrams and partial orders allow the verification of larger systems, many practical timed circuits are still too large to be efficiently analyzed using these techniques alone.

Compositional reasoning and abstraction are essential to verifying large systems. Compositional verification based on assume-guarantee style reasoning explores the inherent modular structure in systems [11]–[15], and it has been applied to the verification of timed circuits [16]. Compositional verification makes assumptions about the environment with which the system interacts, then checks these assumptions later. These assumptions are typically generated by hand. Therefore, if the system has complex interactions with its environment, it can be difficult to make accurate assumptions. Abstraction produces the reduced model of a system by abstracting away certain details that are unnecessary when reasoning about the system [17], [18]. In [19], hand abstractions are used for the verification of timed synchronous domino circuits in the guTS design [1]. In both cases, the assumptions and abstractions are generated by hand, making these techniques difficult to apply except by an expert user. In [20], an automated approach is described to generate the assumptions for compositional verification. This approach starts with a set of the weakest assumptions for a component, and iteratively refines these assumptions. Although the approach guarantees that the iteration terminates, it is not clear how efficient the approach would be in terms of iterations necessary to generate a set of assumptions to prove the properties. Also, this approach can only handle safety properties. In [21], a hierarchical approach similar to that in [22] is presented. In this approach, an abstraction for each module in a system is found and verification is applied to the composition of those abstractions. In [23], a constraint-oriented proof methodology is applied to verify infinite systems. Constraints on infinite systems are broken into an infinite number of simple constraints on finite systems, then these constraints are grouped into finite equivalent classes. However, this methodology is not complete in that the reduction of infinite systems is not guaranteed. In [24], a software model-checking method utilizing lazy abstraction is presented to improve performance by adding information during abstraction refinement only when necessary. It would be interesting to see if this method can be adapted to hardware verification. Predicate abstraction has generated a lot

Manuscript received November 18, 2004; revised February 8, 2005. This work was supported by the Semiconductor Research Corporation (SRC) Contract 2002-TJ-1024, National Science Foundation (NSF) Japan Program Award INT-0087281, and Japan Society for the Promotion of Science (JSPS) Joint Research Projects. This paper was recommended by Associate Editor J. H. Kukula.

H. Zheng is with the Computer Science and Engineering Department, University of South Florida, Tampa, FL 33620 USA.

C. J. Myers is with the Electrical and Computer Engineering Department, University of Utah, Salt Lake City, UT 84112 USA.

D. Walter and S. Little are with the School of Computing, University of Utah, Salt Lake City, UT 84112 USA.

T. Yoneda is with the National Institute of Informatics in Tokyo, Japan.

Digital Object Identifier 10.1109/TCAD.2005.854638

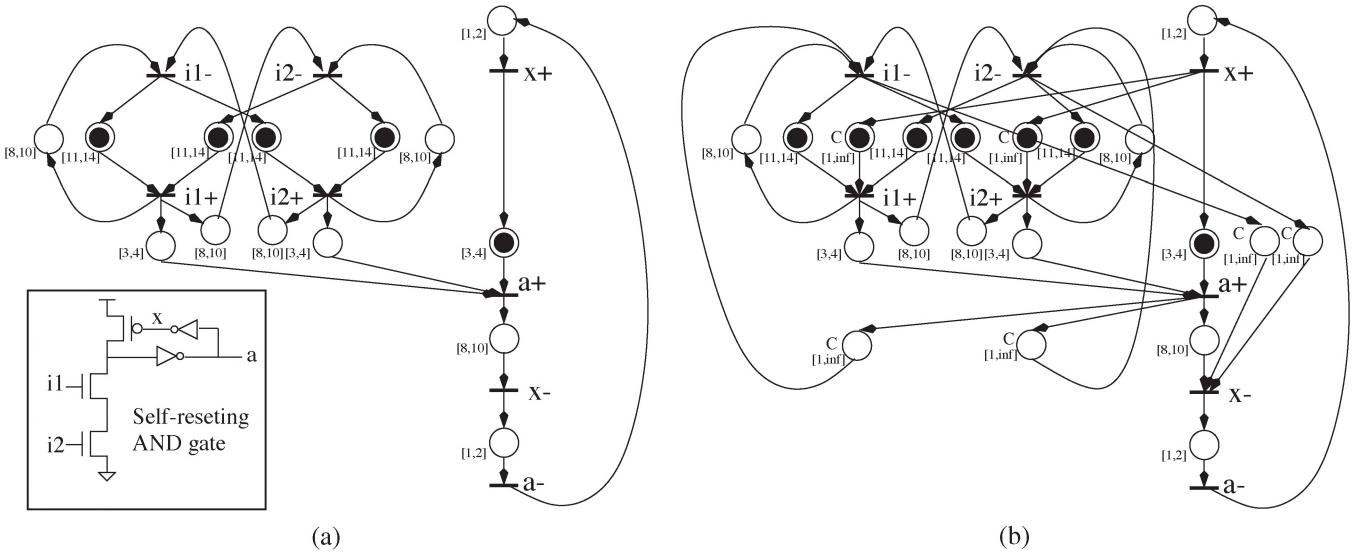

Fig. 1. (a) TPN for a self-resetting AND gate; (b) TPN including timing constraints.

of interest [25]–[27]. First described by Graf and Saïdi [25], predicate abstraction is a technique that combines theorem proving and model checking automatically by mapping an unbounded concrete system into an abstract finite state system where the states correspond to truth assignments to a set of predicates. Recently, predicate abstraction has been applied to the verification of timed systems [28]. It would be interesting to see how predicate abstraction can be combined with our method to further improve performance.

A method that combines compositional reasoning and abstraction to reduce the cost of timing verification is presented in [29]. By utilizing the inherent modular structure in hardware designs, each module in a design is verified individually. Before verification, information in the environment that is irrelevant to reasoning about the module being verified is abstracted away. Then, that module is verified with its abstracted environment. While this work has been shown to verify larger circuits, it cannot be applied to flat designs or ones where the size of individual modules is beyond the capacity of the timing-verification tool. In these cases, the module must first be decomposed by hand into smaller submodules.

This paper addresses this problem by dividing the verification problem as directed by the failure model rather than by the module interface boundaries. Timing verification is utilized to show that several different failure conditions cannot arise. This paper proposes to decompose the verification problem into several subproblems in which each of the failure conditions is checked individually. In this form of problem decomposition, any information in a model irrelevant to a given failure condition is a candidate for abstraction. As shown later in the paper, each failure condition in our model involves only a very small amount of information, which allows abstraction to produce a substantial reduction in the size of the verification problem. This work extends the method in [29] to allow for abstraction independent of the hierarchical structure of the design. In other words, the method can now be applied to flat designs or designs that include large modules. This is desirable in that it eliminates the requirement of functionally unnatural partitioning for the underlying timing-verification tool and the time spent in searching for such a partition. It also avoids errors incurred during decomposition. The decomposition and abstraction method described in this paper is proved to never produce a false-positive verification result. Although the method can produce a false-negative result, this paper describes an algorithm that examines the abstract error trace either to determine a concrete error trace or report that the result is a false negative. Finally, this paper demonstrates the effectiveness of this method by its application to several large-scale timed-circuit designs.

II. Timed Petri Nets (TPNs)

Our method uses TPNs [30] to specify timed-circuit behaviors. 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 nonnegative rational and nonnegative real numbers, respectively. A W-labeled one-safe TPN is a directed bipartite digraph described by the tuple $N = (T, P, F, M_0, \mathsf{l}, \mathsf{u}, C, L)$, where 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; $\mathsf{l} : P \rightarrow \mathbb{Q}^+$ is the lower timing-bound function; $\mathsf{u} : P \rightarrow \mathbb{Q}^+ \cup \{\infty\}$ is the upper timing-bound function; $C \subseteq P$ is the set of constraint places; and $L : T \rightarrow (W \times \{+, -\})$ is the labeling function.

A transistor diagram for a self-resetting AND gate with specific timing information and a TPN representing its behavior and that of its environment are shown in Fig. 1(a). A self-resetting AND gate receives a pulse on input $i1$ and $i2$ and generates a pulse on output a. Intuitively, the TPN shows that $i1$ and $i2$ go high after eleven to fourteen time units. After three to four more time units, a goes high. Also, after eight to ten time units, $i1$ and $i2$ go low. The internal signal x goes low eight

to ten time units after a goes high. This, in turn, resets a one to two time units later, which sets x high after one to two more time units, returning the circuit to its initial state.

The self-resetting AND gate is correct if it satisfies the following requirements: 1) **hold time**: the signal a must go high one time unit before either $i1$ or $i2$ goes low; 2) **short circuit**: the signal x must not go low until one time unit after both $i1$ and $i2$ have gone low, and $i1$ and $i2$ must not go high again until one time unit after x has gone high. Constraint places are used to specify these types of ordering and timing requirements between transitions. The constraint places marked with a "C" in Fig. 1(b) are used to check the above requirements. For example, the hold-time requirement is checked using constraint places in the postset of $a+$.

The remainder of this section describes the formal semantics of TPNs in more detail. The state of a Petri net is a marking M, which is the set of places that hold tokens. With every transition $t \in T$, its associated preset is $\bullet t = \{p \in P \mid (p,t) \in F\}$. The place set of a transition is the restriction of places in its preset to ordinary (not constraint) places, i.e., $\circ t = \bullet t - C$. For a transition $t \in T$, its associated postset is $t\bullet = \{p \in P \mid (t,p) \in F\}$. Note that the preset and postset for places are defined in a similar manner. A transition is enabled in M if $\circ t \subseteq M$. The set of transitions enabled in M is denoted by $X(M)$. Our method requires correct nets to be one safe (i.e., each place is allowed to contain no more than one token).[1]

The state of a TPN is a pair (M, D) where M is the current marking and $D : P \rightarrow \mathbb{R}^+$ is a clock-assignment function assigning nonnegative real numbers to places. For every place p, the value $D(p)$ is the value of a clock associated with p denoting its age. There are two operations on clocks: advance and reset. For some nonnegative real number $d \in \mathbb{R}^+$, $D + d$ advances the clock for every $p \in P$ to the value $D(p) + d$. For some subset of places $\hat{P} \subseteq P$, $[\hat{P} \mapsto 0]D$ resets the clock for every place in $\hat{P}$ to zero, and agrees with D for every place in $P - \hat{P}$. The initial clock assignment D_0 is defined such that every clock is zero. The initial state of a TPN is the pair (M_0, D_0).

The state of a TPN can change by firing a transition or advancing time. To fire a transition t at (M, D), in addition to t being enabled, D must satisfy the timing constraints defined by l and u. A transition is time enabled if it is enabled and: 1) the clock for each place in its place set is above its lower bound [i.e., $\forall p \in \circ t.\ D(p) \geq \mathsf{l}(p)$]; and 2) there exists a clock for a place in its place set that is below its upper bound [i.e., $\exists p' \in \circ t.\ D(p') \leq \mathsf{u}(p')$]. Firing a time-enabled transition t from (M, D) creates the new state (M', D'), denoted by $(M, D)[t\rangle(M', D')$, where $M' = (M - \bullet t) \cup t\bullet$ and $D' = [t\bullet \mapsto 0]D$.

The state of a TPN can also change by advancing time. Advancing time only affects the clock-assignment function in the state pair. Advancing time by a delay $d \in \mathbb{R}^+$ in (M, D) creates a new state (M, D'), denoted by $(M, D)[d\rangle(M, D')$, where $D' = D + d$. Time is not allowed to advance beyond the point where it would disable a time-enabled transition. The maximum delay advancement, $d \in \mathbb{R}^+$, at state (M, D) is

$$d_{\max}(M, D) = \min_{t \in X(M)} \left(\max_{p \in \circ t} (\mathsf{u}(p) - D(p)) \right).$$

After advancing time by the maximum delay, a transition either remains not time enabled, becomes time enabled, or is already time enabled and remains so.

For example, Fig. 1(a) shows the TPN with the initial marking for the self-resetting AND gate. Transitions $i1+$ and $i2+$ are enabled in the initial marking, while $a+$ is not because two places in its preset do not have tokens. After four time units, the clock between $x+$ and $a+$ expires. This simply means that firing $a+$ is no longer constrained by this place because the other two places have not yet acquired tokens. After eleven time units, $i1+$ and $i2+$ become time enabled because the clocks for the places in their presets exceed their lower bounds. Before fourteen time units elapse, both $i1+$ and $i2+$ must fire. After firing both $i1+$ and $i2+$, the tokens are removed from their presets and new tokens and clocks are introduced in their postsets. At this point, $a+$ now has all the tokens it needs to fire, and $a+$ fires three to four time units later after $i1+$ and $i2+$.

III. Timed Trace Theory

This paper uses trace theory to define the semantics for TPNs. Trace theory has been used for the verification of both speed-independent [22] and timed circuits [10]. Given a TPN N, a trace of N, $x = (e_1, e_2, \ldots)$, is a sequence of transition-time pairs, where $e_i = (t_i, \tau_i)$. The time, τ_i is an absolute time stamp for transition t_i. The trace $(e_1, \ldots, e_n)$ is a valid trace if there exists a sequence of states $(s_0, s_1, \ldots, s_n)$ such that for $1 \leq i \leq n$, each $s_i = (M_i, D_i)$ and $d_i = \tau_i - \tau_{i-1}$ (note $\tau_0 = 0$):

1) $0 \leq d_i \leq d_{\max}(s_{i-1})$;
2) $(M_{i-1}, D_{i-1})[d_i\rangle(M_{i-1}, D')$;
3) t_i is time enabled in (M_{i-1}, D');
4) $(M_{i-1}, D')[t_i\rangle(M_i, D_i)$.

The set of all possible valid traces for a TPN N starting from the initial state M_0, D_0 is denoted by $\mathcal{P}(N)$. Although this set is infinite, there exist numerous algorithms for timed state space exploration of Petri-net models that represent this infinite set of traces using a finite set of equivalent state classes (see [31]–[34] for example).

The delete function, $\mathbf{del}(\mathcal{D})(x)$, removes all transition-time pairs of a trace $x = (e_1, e_2, \ldots)$ whose transitions are in $\mathcal{D}$. More formally, if $x \neq \epsilon$ (i.e., the empty trace), then

$$\mathbf{del}(\mathcal{D})(x) = \begin{cases} (e_1, y), & \text{if } t_1 \notin \mathcal{D} \\ (y), & \text{if } t_1 \in \mathcal{D} \end{cases}$$

where $y = \mathbf{del}(\mathcal{D})(e_2, e_3, \ldots)$ and $e_i = (t_i, \tau_i)$. If $x = \epsilon$, then $\mathbf{del}(\mathcal{D})(x) = \{\epsilon\}$. This function is extended naturally to sets of traces.

The set of valid traces in a TPN is divided into those that are successes and those that are failures. There are three types of failures that are considered in this paper: safety, complement, and constraint failures. A valid trace is a safety failure if in

[1] As described later, our analysis method checks for violations of the one-safe property during analysis, and when such a violation is detected, a failure is reported and analysis ceases.

firing the trace the marking update tries to add to the new marking a place that already exists in the current marking. The one-safe requirement of TPNs is common for timed state space exploration algorithms. An unsafe net (i.e., one that is not one safe) typically indicates a problem with the design. Note that this definition of safety is on the reachable state space, so while the TPN may not be structurally safe in an untimed sense, a failure is only reported when a marking is actually reached that violates the safety property. A valid trace is a complement failure on wire w if there exist two rising (falling) transitions on w without a falling (rising) transition in between. Complement failures are also a common modeling error typically caused by the designer while creating the circuit description when the set and reset phases of a signal are similar. A valid trace is a constraint failure if it contains a transition or time progress that could not have occurred if constraint places are taken into account in the definition of enabledness. Constraints are used to indicate required ordering and timing relationships, and they are the key tool for describing necessary properties of a circuit such as hold time, short-circuit avoidance, etc. There are three failure conditions for constraints. First, a transition having a constraint place in its preset is taken while the constraint place is not marked or has not been marked long enough. This indicates either a desired ordering of signals that is violated, or a minimum time separation between signals does not hold. The second part of the definition indicates when a token stays in a constraint place beyond its upper bound. This is used to set maximum time separations between transitions. The third part states the condition when a circuit deadlocks while a constraint place is marked. It is used to check that a desired behavior occurs before the circuit deadlocks.

In our method, the function $\mathbf{fail}(N, W', C')$ is introduced to take a TPN N, a set of wires W', and a set of constraint places C', and returns a subset of $\mathcal{P}(N)$ that are either safety failures, complement failures on W', or constraint failures involving places in C'. In other words, a valid trace $((t_1, \tau_1), \ldots, (t_n, \tau_n))$ is returned by $\mathbf{fail}(N, W', C')$ if, for its corresponding state sequence $(s_0, s_1, \ldots, s_n)$, one of the following conditions is true.

1) **Safety failure**: there exist $s_{i-1} = (M_{i-1}, D_{i-1})$ and pair (t_i, τ_i), where $(M_{i-1} - {\bullet t_i}) \cap t_i{\bullet} \neq \emptyset$.
2) **Complement failure**: there exist a $w \in W'$ and pairs (t_i, τ_i) and (t_k, τ_k) such that the following is true:
   a) $i < k$;
   b) $(L(t_i) = L(t_k) = w+) \vee (L(t_i) = L(t_k) = w-)$; and
   c) $\forall j \cdot i < j < k \wedge (L(t_i) = w+ \Longrightarrow L(t_j) \neq w-) \wedge (L(t_i) = w- \Longrightarrow L(t_j) \neq w+)$.
3) **Constraint failure**: there exist a $c \in C'$, $s_{i-1} = (M_{i-1}, D_{i-1})$, and $d_i = \tau_i - \tau_{i-1}$, such that one of the three following conditions hold:
   a) $c \in {\bullet t_i} \wedge ((c \notin M_{i-1}) \vee (D_{i-1}(c) + d_i < \mathrm{l}(c)))$;
   b) $c \in M_{i-1} \wedge D_{i-1}(c) + d_i > \mathrm{u}(c)$; or
   c) $X(M_i) = \emptyset \wedge c \in M_i$.

If $W' = \emptyset$ and $C' = \emptyset$, $\mathbf{fail}(N, \emptyset, \emptyset)$ only returns traces that would cause safety failures in N. Similarly, $\mathbf{fail}(N, W', \emptyset)$ only returns traces that would cause safety failures or complement failures on signals in W', while $\mathbf{fail}(N, \emptyset, C')$ only returns traces that would cause safety failures or constraint failures on constraint places in C'.

IV. Safe Transformations

In [29], we introduced the notion of safe transformations that can be used to reduce the size and complexity of TPN specifications. In particular, a transformation $\pi_i(N)$ returns a new net N', and it is defined to be safe when the TPN resulting from this transformation satisfies the following two properties:

$$\mathcal{P}(N') \supseteq \mathbf{del}(T - T')(\mathcal{P}(N)) \tag{1}$$

$$\mathbf{fail}(N', \emptyset, \emptyset) \supseteq \mathbf{del}(T - T')(\mathbf{fail}(N, \emptyset, \emptyset)) \tag{2}$$

where T' is the set of transitions in N'. In other words, a net produced by a safe transformation produces a superset of the timed traces produced by the original TPN when any abstracted transition is deleted from these traces, and the transformation does not hide a safety failure of the net. As shown in the following lemma, the application of a sequence of safe transformations is also a safe transformation.

Lemma 4.1: If $\pi_i(N)$ and $\pi_j(N)$ are safe transformations, then so is $\pi_j(\pi_i(N))$.

Proof: Assume $N' = \pi_i(N)$ and $N'' = \pi_j(N')$. From the definition of a safe transformation, we have

$$\mathcal{P}(N') \supseteq \mathbf{del}(T - T')(\mathcal{P}(N))$$
$$\mathcal{P}(N'') \supseteq \mathbf{del}(T' - T'')(\mathcal{P}(N')).$$

Combining these two equations, we get

$$\mathcal{P}(N'') \supseteq \mathbf{del}(T' - T'')(\mathbf{del}(T - T')(\mathcal{P}(N))) = \mathbf{del}(T - T'')(\mathcal{P}(N)).$$

This proves the first half of the definition of a safe transformation. The second half [i.e., $\mathbf{fail}(N'', \emptyset, \emptyset) \supseteq \mathbf{del}(T - T'')(\mathbf{fail}(N, \emptyset, \emptyset))$] is proven similarly. ■

In the rest of this section, we present some TPN reductions that satisfy the safe-transformation properties. Murata [35] presents several transformations on untimed Petri nets that preserve the safety properties of the original net. We have extended these transformations and developed others for TPNs [29]. Two example safe transformations are shown in Figs. 2 and 3. More information on safe transformations can be found in [29]. If our method is working on a net N and finds a portion of the net that resembles that shown in Fig. 2(a), and t is a transition that can be abstracted, it can transform N to a new net N' in which t has been removed, as shown in Fig. 2(b), where the timing bounds have been combined, as shown, to preserve the timing behavior. Note that, although shown with only two places in the preset of t, this transformation is valid for any number of places in the preset of t as long as there is only one place in the postset of t. While the places in the preset of t can have any number of transitions in their presets, they must only have transition t in their postset [i.e., $({\bullet t}){\bullet} = \{t\}$]. Similarly, the place in the postset of t can have any number of transitions in its postset, but it must only have transition t in its preset [i.e., ${\bullet}(t{\bullet}) = \{t\}$]. In a similar fashion, if transition t has

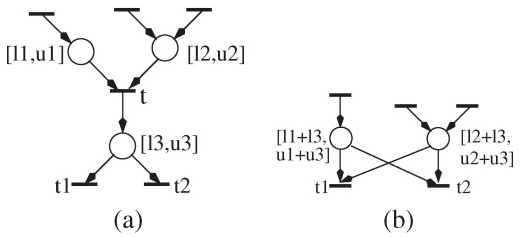

Fig. 2. Safe transformation 1.

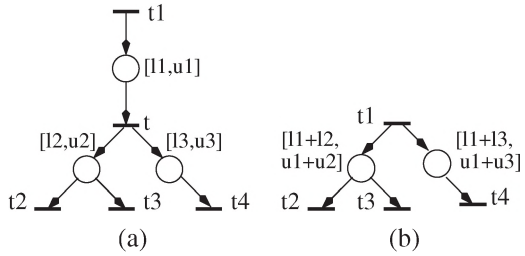

Fig. 3. Safe transformation 2.

only a single place in its preset and satisfies similar restrictions, it can again be removed, as shown in Fig. 3. The application of these transformations is polynomial in the size of the net.

V. Failure-Directed Abstraction

A timed-circuit description is defined to be correct if $\mathbf{fail}(N, W, C) = \emptyset$. This section presents an approach to proving $\mathbf{fail}(N, W, C) = \emptyset$ by showing that:

1) $\mathbf{fail}(N, \emptyset, \emptyset) = \emptyset$;
2) $\forall w \in W.\ \mathbf{fail}(N, \{w\}, \emptyset) = \emptyset$;
3) $\forall c \in C.\ \mathbf{fail}(N, \emptyset, \{c\}) = \emptyset$.

Now, instead of one verification run, our method performs $1 + |W| + |C|$ runs. Note that $\mathbf{fail}(N, \emptyset, \emptyset)$ checks safety properties explicitly, but when $|W| + |C| \geq 1$, this does not need to be done as a separate step since it is checked implicitly during the other checks.

At this point, each run is nearly as complex as the original run, but for each subproblem, not all transitions in N are required to determine if failure traces exist. Therefore, in the second step, our method constructs a set of transitions that can potentially be abstracted safely without causing failures to be missed. The function $\mathcal{D}(N, W', C')$ takes a set of wires $(W' \subseteq W)$ and a set of constraint places $(C' \subseteq C)$, and it returns the following set:

$$\{t \in T | (\forall w \in W'.\ L(t) \neq w+ \wedge L(t) \neq w-) \wedge (\forall c \in C'.\ t \notin {\bullet c} \cup c{\bullet})\}.$$

Intuitively, $\mathcal{D}(N, W', C')$ returns a set of transitions in N such that they are not transitions on wires in W', and not in the preset and postset of constraint places in C'. For example, let N denote the TPN shown in Fig. 1, and $p1$ and $p2$ denote the constraint places in the postset of $a+$. Then, $\mathcal{D}(N, \{i1, i2\}, \{p1, p2\}) = \{x+, x-, a-\}$.

Finally, the third step of our method is to apply safe transformations to the net to remove the transitions returned by $\mathcal{D}(N, W', C')$ and the related places, whenever possible. Note that not all transitions returned by $\mathcal{D}(N, W', C')$ can be safely removed, but only those that can be removed via safe transformations. We define a function $\mathbf{abs}(N, W'', C'')$ that takes a TPN N, a set of wires W'', and a set of constraint places C'', and applies a sequence of safe transformations to remove, when possible, transitions in $\mathcal{D}(N, W'', C'')$ from N to obtain a new TPN N'. The safe transformations used are restricted such that $T - T' \subseteq \mathcal{D}(N, W'', C'')$ and for all $c \in C''$, c is in the initial marking of the new net M_0' if and only if c is in the initial marking of the original net M_0. The result after applying this function to a net is typically a net that is substantially simpler, and thus, results in a much smaller state space. The main theorem can now be presented.

Theorem 5.1: Let N be a TPN. $\mathbf{fail}(N, W, C) = \emptyset$ if the following three conditions are true:

1) $\mathbf{fail}(\mathbf{abs}(N, \emptyset, \emptyset), \emptyset, \emptyset) = \emptyset$;
2) $\forall w \in W.\ \mathbf{fail}(\mathbf{abs}(N, \{w\}, \emptyset), \{w\}, \emptyset) = \emptyset$;
3) $\forall c \in C.\ \mathbf{fail}(\mathbf{abs}(N, \emptyset, \{c\}), \emptyset, \{c\}) = \emptyset$.

Proof: We break up this proof into three cases.

Case 1) **(Safety failures)** Assume there is a trace x that causes a safety failure in N, and that N' is the TPN returned by the function $\mathbf{abs}(N, \emptyset, \emptyset)$. Since $x \in \mathbf{fail}(N, \emptyset, \emptyset)$, there must also exist a trace $y = \mathbf{del}(T - T')(x)$ such that $y \in \mathbf{del}(T - T')(\mathbf{fail}(N, \emptyset, \emptyset))$. According to property (2), $y \in \mathbf{fail}(N', \emptyset, \emptyset)$. Therefore, a safety failure is detected on the abstracted net.

Case 2) **(Complement failures)** Assume there is a trace x that causes a complement failure on signal w in N, and that N' is the TPN returned by the function $\mathbf{abs}(N, \{w\}, \emptyset)$. Since $x \in \mathcal{P}(N)$, there must also exist a trace $y = \mathbf{del}(T - T')(x)$ such that $y \in \mathbf{del}(T - T')(\mathcal{P}(N))$. From property (1), we know safe transformations do not hide any timed traces, so y must also be in $\mathcal{P}(N')$. From the definition of complement failure, there exist two transitions t_i and t_k on signal w that create the complement failure. In fact, only transitions on w are required to show if a trace is or is not a complement failure. By the definition of $\mathbf{abs}$, the trace y must include all transitions on signal w in trace x with some additional transitions from $\mathcal{D}(N, \{w\}, \emptyset)$ that could not be abstracted. Since a complement failure is detected by only examining those transitions on signal w and x is a complement failure, y is also a complement failure because removing transitions not on w does not change whether a trace is a complement failure or not.

Case 3) **(Constraint failures)** Assume there is a trace x that causes a constraint failure on constraint place c in N, and that N' is the TPN returned by the function $\mathbf{abs}(N, \emptyset, \{c\})$. Since $x \in \mathcal{P}(N)$, there must also exist a trace $y = \mathbf{del}(T - T')(x)$ such that $y \in \mathbf{del}(T - T')(\mathcal{P}(N))$. This trace consists of all transitions in ${\bullet c} \cup c{\bullet}$ plus some additional transitions from $\mathcal{D}(N, \emptyset, \{c\})$ that could not be

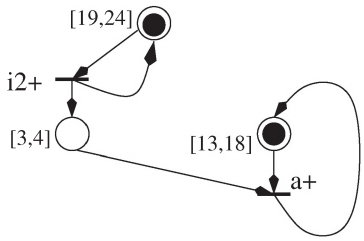

Fig. 4. TPN for checking safety.

abstracted. Traces x and y agree on the timing of all transitions in $\bullet c \cup c\bullet$. There are three types of constraint failure defined in Section III. For type 3a), since all transitions in $\bullet c$ are preserved as is the initial marking of c, the value of the predicate $c \notin M_{i-1}$ is preserved at the time of firing t_i (a transition that is also preserved). The value of $D_{i-1}(c)$ is also preserved for this reason. Therefore, if trace x violates 3a), so does trace y. If trace x violates type 3b), this means a transition from $\bullet c$ fired to put a token into c, then either a transition in $c\bullet$ fired but fired too late (in this type, it is preserved in y, so it is okay) or a transition outside $\bullet c \cup c\bullet$ fired to cause the failure. In this type, that transition is not necessarily preserved in y. However, either there exists another transition in y that causes the upper bound violation, or if the trace is finite and ends with no transitions being enabled, y is a failure due to type 3c). In either case, the failure is found examining y. Finally, type 3c) is preserved from x to y as in both traces end with no enabled transitions and the constraint place in the marking.

Therefore, if there exists a failure trace in the concrete description, it is found by an analysis of one of the abstract descriptions. ■

Suppose that we would like to check if the TPN for the self-resetting AND gate shown in Fig. 1 has safety failures. First, all constraint places in the TPN are removed because they are only needed for checking constraint failures. Since only safety failures are checked, all transitions in the TPN are candidates for removal. After applying safe transformations such as those described earlier, as well as those from [29], the TPN in Fig. 1 is reduced to the one shown in Fig. 4. A timing analysis of this net shows that $i2+$ fires after 19 to 24 time units, followed by $a+$ after three to four time units, which enables $i2+$ to fire again after another 19 to 24 time units. Notice that the self-loop on $a+$ is never constraining.

Note that the reduced TPN contains complement failures on signals $i2$ and a, but they are ignored during this particular verification run because the reduced TPN is generated only for checking safety failures. Separate verification runs are required to check complement and constraint failures as described at the beginning of this section and formalized in Theorem 5.1. In particular, this example has four wires, and its TPN contains six constraint places. The decomposition method described in this paper would verify this example using 11 verification runs, one for each wire and constraint place, and another one for checking safety failures. This example illustrates how failure-directed

```
find_concrete_trace(N, T', x)
  s = (M_0, D_0)
  (t, τ) = head(x)
  x = tail(x)
  x' = ε
  do
    if ((t, τ) is time-enabled in s) then
      E = dependent(t, s, T')
      push(s, x, x', E − {(t, τ)})
      (t', τ') = (t, τ)
      (t, τ) = head(x)
      x = tail(x)
    else
      Nec = necessary_set(t, s, T', ∅)
      E = ∅
      foreach t' ∈ Nec
        E = E ∪ dependent(t', s, T')
      if (E = ∅) then
        if (stack not empty) then
          pop(s, x, x', E)
        else return false negative
      (t', τ') = choose_one(E)
      E = E − {(t', τ')}
      if (E ≠ ∅) then push(s, x, x', E)
    s = fire((t', τ'), s)
    x' = (x', (t', τ'))
  while (x is not empty)
  return x'
```

Fig. 5. Algorithm to find concrete trace.

abstraction is applied to reduce net complexity, but breaking the verification into 11 runs is clearly overkill for such a small example. However, for large examples, as shown in Section VII, this type of decomposition can improve the overall verification time significantly and allow the verification of designs that cannot be handled previously. In particular, the state space that must be explored for each subproblem is usually exponentially smaller than the original due to the net reductions, and each subproblem is constructed with an algorithm that is polynomial in the size of the original net.

VI. Handling False Negatives

The verification method just described is conservative in that false negatives are possible and false positives never occur. Consider again the transformation shown in Fig. 3. In this case, the summing of the timing bounds as shown in the figure may actually result in new timed traces. For example, in the new net, the trace $((t_1, 0), (t_2, l_1 + l_2), (t_4, u_1 + u_3))$ is possible, while this is not possible in the original net. It does, however, produce all the timed traces of the original net, so it is a safe transformation. If this extra timing introduces new timed traces and one of those introduced traces causes a failure, this failure is a false negative. Therefore, when an error trace is reported from an analysis of the abstracted net, it is not known whether this is a real error trace or a false one. Also, it is difficult for a designer to analyze the error trace to find the problem as it only includes the transitions that have not been abstracted.

To address both of these problems, this paper introduces the algorithm shown in Fig. 5. This algorithm uses the abstract error trace to perform a guided simulation of the original TPN to find a concrete error trace. This is done by attempting to fire

```
necessary_set(t, s, T', T_D)
  if (t ∈ T_D) then return ∅
  if (t ∈ enabled(s)) then
    if (t ∈ time-enabled(s)) then return {t}
    else return { choose_one(time-enabled(s) − T') }
  res = ∅
  foreach (p ∈ •t − μ(s))
    temp = ∅
    foreach (t' ∈ •p − T')
      temp = temp ∪ necessary_set(t', s, T', T_D ∪ {t})
    if ((res = ∅) or (| temp | ≤ | res |)) then
      res = temp
  return res
```

Fig. 6. Algorithm to find a necessary set.

```
dependent(t, s, T')
  R = C = {t}
  while(true)
    C' = ∅
    foreach(x ∈ C)
      foreach(y ∈ conflict(x) − (T' ∪ {x}))
        C' = C' ∪ (necessary_set(y, s, T', ∅) − R)
    if (C' = ∅) then return R
    R = R ∪ C'
    C = C'
```

Fig. 7. Algorithm to find a dependent set.

transitions from the abstract error trace, and when one of these transitions is not fireable, it examines the TPN to determine an abstracted transition to fire, which contributes toward the enabling of the next transition in the abstract error trace. In general, multiple such transitions may exist and the algorithm may need to explore multiple paths to find a valid concrete error trace. When no concrete error trace can be found, it is reported that the abstract error trace is false. In this case, abstraction can be performed again using a smaller subset of transformations by removing transformations known to add behavior. This process can be repeated until all behavior-adding transformations are removed from the subset of transformations used. While in the worst case, disallowing transformations that add timed traces can result in a flat verification, we have not seen this happen in practice.

The guided simulation used in the algorithm for finding a concrete trace is based upon methods developed for partial-order state-space exploration [10]. In particular, the **find_concrete_trace** algorithm calls two functions used during a typical partial-order state exploration, **necessary_set** and **dependent**. The **necessary_set** algorithm in Fig. 6 is used to determine which transitions must fire before transition t can fire. This algorithm takes the transition to fire t, a timed state s, a set of transitions that should not be fired T', and the set of transitions visited so far T_D. The **necessary_set** algorithm proceeds in the following manner. First, it checks if t is in the set of visited transitions T_D. If it is, a cycle is detected, and **necessary_set** returns the empty set. If t is a time-enabled transition, then it returns t. If t is enabled but not time enabled, then there must exist some other time-enabled transition that must fire first. Therefore, one time-enabled transition is chosen at random to fire to allow time to move forward. If t is not enabled, then the algorithm must look backward in the Petri net to determine which transitions must fire in order to enable t. This is done by finding all of the unmarked places p in the preset of t and then calling **necessary_set** on each transition t' that is in the preset of p and not a member of the set of transitions that should not be fired T'. The result of this operation for each p forms the set of transitions that are necessary to fire in order to allow t to fire. The smallest of these sets is returned.

The **dependent** algorithm shown in Fig. 7 is used to find a set of transitions that must be interleaved. Transitions must be interleaved because they are in conflict. Transitions conflict when they share a common preset place (i.e., conflict$(t) = \{t' \in T \mid \bullet t \cap \bullet t' \neq \emptyset\}$). The **dependent** algorithm takes a seed transition t, a timed state s, and a set of transitions that should not be fired T'. The dependent-set calculation begins with an initial dependent set consisting of just the transition t. The algorithm then looks for additional transitions that conflict with those already in the dependent set. These transitions may not yet be enabled, so this algorithm uses the **necessary_set** algorithm to find those transition firings that would lead to the enabling of the conflicting transition. Each time through the loop, newly found transitions are added to the set and this loop continues until no new transitions are found. The set of transitions dependent on t are then returned.

The **find_concrete_trace** algorithm takes as input the initial Petri net before abstraction N, the set of transitions in the abstracted net T', and an abstract trace x, and it proceeds in the following manner. First, it sets the current state s to the initial state (M_0, D_0). The first item (t, τ) is removed from the abstract error trace x. If t is time enabled in the current state, then the dependent-set information is calculated and pushed on the stack for backtracking purposes. Next, t is fired and added to the concrete error trace x'. A new t is selected from the abstract error trace and the process continues. If t is not time enabled, then the set of transitions necessary for it to become time enabled is calculated. For each of these necessary transitions, the set of transitions that must be interleaved with each of them is calculated and added to the set of necessary transitions. If this set E is empty, then we have reached a dead path. If more possible interleavings are on the stack, then the algorithm backs up to the point at which the last choice was made. If the stack is empty, then a false negative has been found. If the set E is not empty, then a transition from the set is selected, and if any transitions remain, they are pushed on the stack for possible backtracking. The selected transition is fired and added to the concrete trace and control loops back to the beginning.

The main idea behind the concrete-trace algorithm is to use the abstract trace to guide a search through the state space of the flat design, using a partial-order reduction-based reachability algorithm, to confirm or refute the existence of a concrete trace containing the abstract trace. Since a reachability algorithm is used, it is possible that searching for a concrete error trace could result in a full state-space exploration of the flat design. We have, however, not seen this behavior in practice. As with partial-order methods, this trace-generation method works well with designs that have a high degree of concurrence. In designs with more conflict than concurrence, the chance of seeing worst case behavior increases.

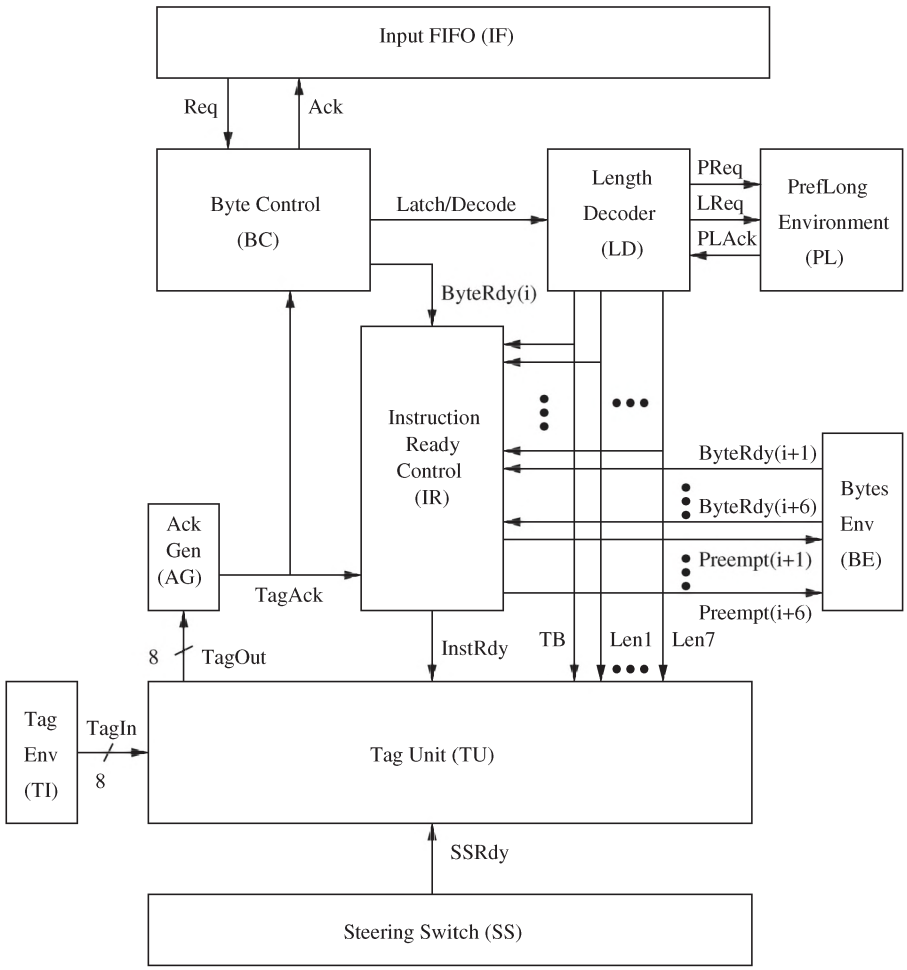

Fig. 8. Block diagram for RAPPID circuit.

TABLE I
EXPERIMENTAL RESULTS

	Runs	$\|T\|$	Average $\|T\| - \|D\|$	Average $\|T'\|$	Reduction Time (sec)	Max States	Verification Time (sec)	Total Time (sec)
RAPPID	50	114	2	60	7.16	1835	27.40	45.13
TITAC2	63	231	2	26	14.60	112	1.45	83.88
IIR1	129	331	3	11	59.61	49	0.84	429.05
IIR2	133	323	3	11	60.07	29	0.86	429.71
FIR1	296	781	3	12	673.32	105	1.72	8349.87
FIR2	296	757	3	11	720.05	57	1.99	8776.39

In the self-resetting AND gate example shown in Fig. 1, two of the eleven verification runs fail initially. The two runs that fail are for the two constraint places in the postset of $a+$. In both cases, no concrete error trace is found, so these are false-negative results. After disallowing one very conservative transformation that removes self-loops, these two verification runs succeed. In the experimental results described in the next section, only one false negative is encountered.

VII. EXPERIMENTAL RESULTS

We have incorporated the method described in this paper into the compiler front end of the ATACS tool [36], and we have applied it to several examples. The ATACS tool can perform flat verification, modular verification [29], and the new failure-directed method. In the following experiments, the flat, modular, and failure-directed approaches use the same explicit-state reachability analysis engine and parameter settings [37]. All results are obtained on a 1.7-GHz Pentium M with 1 GB of memory.

The first is Intel's RAPPID circuit, which is a fully asynchronous instruction-length decoder for the Pentium II 32-bit MultiMedia eXtensions (MMX) instruction set [2]. In this instruction set, each instruction can be from 1- to 15-B long, depending on a large number of factors. In order to allow concurrent execution of instructions, it is necessary to rapidly determine the positions of each instruction in a cache line. Instruction-length decoding was a critical performance bottleneck in the Pentium II architecture at the time when RAPPID was being designed. The RAPPID circuit is shown to perform,

on average, three times faster, while using half the power of the comparable synchronous design. This performance improvement is due, in large part, to the highly timed nature of the circuits in this design. Therefore, the correctness of this design is highly dependent on timing parameters. The block diagram for the portion of the RAPPID design that we verified is depicted in Fig. 8. The TPN description of the RAPPID circuit has 114 transitions on 49 signal wires with no constraint places. Our second example is the line fetch module of TITAC2's instruction cache system [38], which is represented using a TPN with 231 transitions derived from a high-level specification [39]. The final four examples (IIR1, IIR2, FIR1, and FIR2) are timed signal transition graph (STGs) used in [40], which are obtained from high-level specifications for an IIR filter and an FIR filter by doing resource allocation under several resource constraints and generating the corresponding STGs with timing based on a 0.25-μm gate library. The TPN representations for the IIR examples have more than 300 transitions while those for the FIR examples have more than 700 transitions. These examples have constraint places to check the correctness of the resource allocation, i.e., each resource is used only sequentially.

Table I shows the reduction and runtime results for our examples. Column 2 of Table I shows the total number of analysis runs necessary to verify the circuit. Column 3 shows the total number of transitions in the design before transformation, column 4 shows the average number of transitions that would remain if all abstractable transitions were removable, and column 5 shows the actual average number of transitions remaining after transformation. Column 6 shows the amount of time devoted to performing net transformations across all runs of each design. The maximum number of states visited during verification is shown in column 7 and the total amount of time necessary to perform all verification runs is shown in column 8. Finally, column 9 shows the total time for verifying each design.

For all the examples, flat analysis runs out of memory. The modular approach is only applicable for the RAPPID example, since all the other examples do not contain any hierarchy. For the RAPPID example, the modular approach decomposes the verification problem into ten subproblems, one for each module shown in Fig. 8. However, as described in [29], the IR module is too large and has to be further decomposed by hand into seven smaller modules. The final verification time for this hand-decomposed design is reported to be 618.3 s.

The failure-directed approach succeeded in verifying all the examples. Over all examples, only one false negative is found, which is in the complement-failure check for one signal in the RAPPID design. Again, by removing one transformation, this false negative is removed and verification can complete successfully. In the failure-directed approach, the net size is decreased by 95% on average. The time required for these reductions is never more than 15 min total over all the necessary runs. The result of these reductions is that the state space that is explored is always less than 2000 timed states and often significantly fewer, with a total verification time over all runs never exceeding a minute and normally taking just a few seconds. The majority of the time spent is simply parsing the large nets and creating the data structures necessary to represent the original very large Petri net.

VIII. Conclusion and Future Work

This paper describes a new method to deal with state explosion by decomposing the timing-verification problem as directed by the given failure model. This decomposition allows for a significant reduction in the size of the model for each subproblem using an automatic abstraction method based on safe transformations. It no longer requires that a design is properly partitioned for successful verification. This method has been applied to several large timed-circuit designs, most of which could not previously be verified. Overall, this method scales very well in that the size of the individual verification problems are only dependent on the complexity associated with a single signal or a single constraint place. This new method can also be built on top of any reachability analysis algorithm for TPNs, and benefit from any improvement in the underlying analysis algorithm. In particular, our preliminary analysis has shown that combining abstraction with a partial-order-based analysis technique can bring even further improvements.

References

[1] H. P. Hofstee, S. H. Dhong, D. Meltzer, K. J. Nowka, J. A. Silberman, J. L. Burns, S. D. Posluszny, and O. Takahashi, "Designing for a gigahertz," *IEEE Micro*, vol. 18, no. 3, pp. 66–74, May/Jun. 1998.

[2] 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.

[3] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 13, no. 4, pp. 401–424, Apr. 1994.

[4] M. Bozga, O. Maler, A. Pnueli, and S. Yovine, "Some progress in the symbolic verification of timed automata," in *International Conference on Computer-Aided Verification*, ser. LNCS, vol. 1254. London, U. K.: Springer-Verlag, 1997, pp. 179–190.

[5] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard, "Difference decision diagrams," in *Computer Science Logic*. Copenhagen, Denmark: IT Univ. Copenhagen, Sep. 1999.

[6] K. G. Larsen, C. Weise, Y. Wang, and J. Pearson, "Clock difference diagrams," *Nord. J. Comput.*, vol. 6, no. 3, pp. 271–298, 1999.

[7] A. Valmari, "A stubborn attack on state explosion," in *International Conference on Computer-Aided Verification*, ser. LNCS, vol. 531. London, U. K.: Springer-Verlag, Jun. 1990, pp. 156–165.

[8] P. Godefroid, "Using partial orders to improve automatic verification methods," in *International Conference on Computer-Aided Verification*, ser. LNCS, vol. 531. London, U.K.: Springer-Verlag, 1990, pp. 176–185.

[9] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. (1998). "Partial order reductions for timed systems," in *Int. Conf. Concurrency Theory*, Nice, France, pp. 485–500. [Online]. Available: citeseer.nj.nec.com/bengtsson98partial.html

[10] T. Yoneda and H. Ryu, "Timed trace theoretic verification using partial order reduction," in *Proc. Int. Symp. Advanced Research Asynchronous Circuits and Systems*, Barcelona, Spain, Apr. 1999, pp. 108–121.

[11] J. Misra and K. M. Chandy, "Proofs of networks of processes," *IEEE Trans. Softw. Eng.*, vol. SE-7, no. 4, pp. 417–426, Jul. 1981.

[12] C. Jones, "Tentative steps toward a development for interfering programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 4, pp. 596–619, Oct. 1983.

[13] O. Grumberg and D. Long, "Model checking and modular verification," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 843–872, May 1994.

[14] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "You assume, we guarantee: Methodology and case studies," in *Proc. Int. Conf. Computer-Aided Verification*, Vancouver, B.C., Canada, 1998, pp. 440–451.

[15] K. L. Mcmillan, "A methodology for hardware verification using compositional model checking," *Science of Computer Programming*, vol. 37, no. 1–3, pp. 279–309, May 2000.

[16] S. Tasiran and R. K. Brayton, "Stari: A case study in compositional and hierarchical timing verification," in *International Conference on*

Computer-Aided Verification, ser. LNCS, vol. 1254. London, U. K.: Springer-Verlag, 1997, pp. 191–201.
[17] E. Clarke, O. Grumberg, and D. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, Sep. 1994.
[18] D. Dams, R. Gerth, and O. Grumberg, "Abstract interpretation of reactive systems," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 2, pp. 253–291, Mar. 1997.
[19] W. Belluomini and C. J. Myers, "Timed circuit verification using TEL structures," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 1, pp. 129–146, Jan. 2001.
[20] J. M. Jensen, D. Giannakopoulou, and C. S. Pasareanu, "Learning assumptions for compositional verification," in *LNCS*, vol. 2619. Berlin, Germany: Springer-Verlag, 2003, pp. 331–346.
[21] H. E. Jensen, K. G. Larsen, and A. Skou. (2000). "Scaling up uppaal automatic verification of real-time systems using compositionality and abstraction," in *Formal Techniques Real-Time and Fault-Tolerant Systems (FTRTFT)*, Pune, India, pp. 19–30. [Online]. Available: citeseer.nj.nec.com/jensen00scaling.html
[22] D. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*, ser. ACM Distinguished Dissertations. Cambridge, MA: MIT Press, 1989.
[23] K. Larsen, B. Steffen, and C. Weise, "A constraint oriented proof methodology," in *Formal Systems Verification*, ser. LNCS, vol. 1169. Heidelberg, Germany: Springer-Verlag, Nov. 1996, pp. 405–435.
[24] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *29th Symp. Principles Programming Languages*, Portland, OR, Jan. 2002, pp. 58–70.
[25] S. Graf and H. Saïdi, "Construction of abstract state graphs with pvs," in *Conf. Computer Aided Verification*, Haifa, Israel, Jun. 1997, vol. 1254, pp. 72–83.
[26] T. Ball and S. Rajamani, "A model and process for software analysis," Microsoft Research, Redmond, WA, Tech. Rep. 2000-14, Feb. 2000.
[27] ——, "Automatically validating temporal safety properties of interfaces," in *SPIN Workshop*, Toronto, Canada, May 2001, vol. 2057, pp. 103–122.
[28] M. Möller, H. Reuß, and M. Sorea, "Predicate abstraction for dense real-time systems," *Electron. Notes Theor. Comput. Sci.*, vol. 65, no. 6, pp. 1–20, Jun. 2002.
[29] H. Zheng, E. Mercer, and C. Myers, "Modular verification of timed circuits using automatic abstraction," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 22, no. 9, pp. 1138–1153, Sep. 2003.
[30] C. Ramchandani, "Analysis of asynchronous concurrent systems by timed Petri nets," Massachusetts Inst. Technol., Cambridge, MA, Project MAC Tech. Rep. 120, Feb. 1974.
[31] D. Dill, S. Nowick, and R. Sproull, "Specification and automatic verification of self-timed queues," Stanford Univ. Press, Stanford, CA, Tech. Rep. CSL-TR-89-387, Aug. 1989.
[32] T. G. Rokicki and C. J. Myers, "Automatic verification of timed circuits," in *Proc. Int. Conf. Computer Aided Verification*, Stanford, CA, 1994, pp. 468–480.
[33] T. Yoneda and B. Schlingloff, "Efficient verification of parallel real-time systems," in *Formal Methods in System Design*, C. Courcoubetis, Ed. Boston, MA: Kluwer, 1997.
[34] W. Belluomini and C. J. Myers, "Timed state space exploration using posets," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 19, no. 5, pp. 501–520, May 2000.
[35] T. Murata, "Petri nets: Properties, analysis, and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.
[36] C. 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 Design Automation Conf.*, Yokohama, Japan, Feb. 2001, pp. 335–340.
[37] E. Mercer, C. Myers, and T. Yoneda, "Improved POSET timing analysis in timed Petri nets," in *10th Workshop Synthesis and System Integration Mixed Technologies (SASIMI)*. Nara, Japan, Oct. 2001, pp. 151–158.
[38] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya, "TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model," in *Proc. Int. Conf Computer Design: VLSI Computers and Processors*, Austin, TX, Oct. 1997, pp. 288–294.
[39] T. Yoneda and C. Myers, "Synthesizing timed circuits from high level specification languages," Nat. Inst. Informatics, Tokyo, Japan, NII Tech. Rep. NII-2003-003E, 2003.
[40] T. Yoneda, A. Matsumoto, M. Kato, and C. Myers, "High level synthesis of timed asynchronous circuits," in *Proc. Int. Symp. Advanced Research Asynchronous Circuits and Systems*, New York, Mar. 2005, pp. 178–189.

Hao Zheng (M'05) received the M.S. and Ph.D. degrees in electrical engineering from the University of Utah, Salt Lake City, in 1998 and 2001, respectively.

Currently, he is an Assistant Professor in the Computer Science and Engineering Department of the University of South Florida, Tampa. His research interests include the application of formal methods in the computer system design, devising abstraction techniques to improve the capability of model checking, and advanced architectures for low power and high performance.

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

He is an Associate Professor in the Department of Electrical and Computer Engineering, University of Utah, Salt Lake City. He is the author of over 50 technical papers and the textbook *Asynchronous Circuit Design*. He is also a coinventor of 4 patents. His current research interests are algorithms for the computer-aided analysis and design of real-time concurrent systems, analog error control decoders, formal verification, asynchronous circuit design, and modeling of biological networks.

Dr. Myers received a National Science Foundation (NSF) Fellowship in 1991, an NSF CAREER Award in 1996, and a Best Paper Award at Async99.

David Walter (S'05) received the B.S. degrees in computer science and computer engineering from the University of Utah, Salt Lake City, in 2001. He is currently pursuing the Ph.D. degree in computer science at the University of Utah.

His current research interests are in the formal verification of analog and mixed-signal systems.

Scott Little received the B.S. degree in computer engineering in 2003 from the University of Utah, Salt Lake City. He is currently an SRC Fellow working toward the Ph.D. degree in computer science at the University of Utah.

His current research interests include formal verification of embedded systems and analog/mixed-signal circuits.

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

In 1985, he joined the staff of Tokyo Institute of Technology, and he moved to National Institute of Informatics, Tokyo, Japan, in 2002, where he is currently a Professor. He was a Visiting Researcher of Carnegie Mellon University from 1990 to 1991. His research activities currently focus on 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 Information Processing Society of Japan.