2003-09-01

Modular Verification of Timed Circuits Using Automatic Abstraction

Eric G. Mercer
eric_mercer@byu.edu

Chris Myers

See next page for additional authors

Original Publication Citation
Q. Zheng, E. G. Mercer, and C. J. Myers, "Modular Verification of Timed Circuits Using Automatic Abstraction," IEEE Transactions on Computer-Aided Design of Integrated Circuits, 22(9):1138-1153, September, 23.

Authors
Eric G. Mercer, Chris Myers, and Hao Zheng

Modular Verification of Timed Circuits Using Automatic Abstraction

Hao Zheng, Eric Mercer, *Member, IEEE*, and Chris Myers, *Member, IEEE*

***Abstract*—The major barrier that prevents the application of formal verification to large designs is state explosion. This paper presents a new approach for verification of timed circuits using automatic abstraction. This approach partitions the design into modules, each with constrained complexity. Before verification is applied to each individual module, irrelevant information to the behavior of the selected module is abstracted away. This approach converts a verification problem with big exponential complexity to a set of subproblems, each with small exponential complexity. Experimental results are promising in that they indicate that our approach has the potential of completing much faster while using less memory than traditional flat analysis.**

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

I. Introduction

In order to continue to produce circuits of increasing speed, designers are considering aggressive circuit styles such as self-resetting or delayed-reset domino circuits. These circuit styles can achieve a significant improvement in circuit speed as demonstrated by their use in a gigahertz research microprocessor (guTS) at IBM [1]. Designers are also considering asynchronous circuits due to their potential for higher performance and lower power consumption as demonstrated by the RAPPID instruction length decoder designed at Intel [2]. This design was three times faster while using only half the power of the synchronous design. The correctness of these new *timed circuit* styles is highly dependent upon their timing parameters, so extensive timing verification is necessary during the design process. Unfortunately, these new circuit styles cannot be efficiently and accurately verified using traditional static timing analysis methods. This lack of efficient analysis tools is one of the reasons for the lack of mainstream acceptance of these circuit styles.

In [3], a hierarchical approach to verification based on trace theory is proposed for the analysis of speed-independent circuits. In this approach, a model of a circuit at one level is regarded as the implementation of the model at the higher level and as the specification of the model at the lower level. The model at the higher level is more abstract and has fewer implementation details. A circuit is a correct implementation if it conforms to its specification. Trace theory has proved to be an excellent model for verifying circuits, and it is trace theory that this paper utilizes to justify its approach.

In [4] and [5], trace theory is extended with a representation where time is modeled as multiples of a discretization constant. Unfortunately, the state space explodes if the delay ranges are large and the discretization constant is set small enough to ensure exact exploration of the state space. In [6], timed automata are introduced to model the behavior of real-time systems. It provides a simple and general way to annotate state-transition graphs with timing constraints using a finite number of real-valued *clocks*. Although this approach eliminates the need to discretize time, the number of timed states is dependent on the size of the delay ranges and the number of concurrently enabled clocks which can quickly explode for even relatively small systems. Representing possible clock values with convex polygons, or zones, [7] alleviates this problem in practice. The zone based representation is the one used by most modern timing verifiers such as ATACS [8]–[10], VINAS-P [11], ORBITS [12], [13], KRONOS [14], and UPPAAL [15]. One feature common to these tools is that they require state space exploration which can explode even for modest size examples.

There do exist many methods and approaches to address the state explosion problem. In [16] and [17], the state space of a transition system is represented symbolically using Bryant's *ordered binary decision diagram* [18]. The symbolic approach has been shown to be capable of representing systems with more than 10^{20} states. There has been some success at the verification of timed systems using binary decision diagrams [19], [20]. Asynchronous systems consist of concurrent processes without a global synchronizing clock. State explosion is particularly serious for asynchronous systems because all possible interleavings among concurrently executed events need to be explored. A number of techniques have been proposed to minimize the number of interleavings that are explored, including stubborn sets [21], partial orders [22], and unfoldings [23]. There has also been some success at applying partial orders to formal timing verification [11], [24]. Although the approaches described above have been successful in verifying systems with increased sizes, many realistic systems are still too large to be handled.

In practice, circuits often have inherent modular structures. Compositional verification methods based on *assume-guarantee reasoning* [25]–[27], exploit the modular structure of circuits. Verifying a circuit component in this approach necessitates behavioral assumptions on connecting components to

Manuscript received December 7, 2001; revised April 2, 2002 and November 25, 2002. This work was supported in part by National Science Foundation CAREER Award MIP-9625014, in part by Semiconductor Research Corporation Contract 97-DJ-487, Contract 99-TJ-694, and Contract 2002-TJ-1024, and in part by a grant from the Intel Corporation. This paper was recommended by Associate Editor J. H. Kukula.

H. Zheng is with IBM, Essex Junction, VT 05452 USA (e-mail: haoz@us.ibm.com).

E. Mercer is with the Department of Computer Science, Brigham Young University, Provo, UT 84602 USA.

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

Digital Object Identifier 10.1109/TCAD.2003.816214

reduce complexity in the model. The assumptions must later be discharged as part of the correctness proof for connecting the components. Our approach is similar to assume-guarantee reasoning, only it does not necessitate behavioral assumptions on the connecting components. In our approach, their complexities are automatically abstracted through semantics-preserving transformations; thus, we do not generate additional proof obligations for the components.

Abstraction is essential to reasoning about circuits with datapath components. It reduces model complexity by generalizing the actual data values into a reduced set of less detail that preserves key properties in the original set. This mitigates state explosion in the model. The properties that need to be preserved depend on the verification problem, but the problem is often much simpler to solve through the abstracted model. Examples of effective abstraction methods are seen in [28] and [29]. These methods, however, do not address real-time systems such as timed circuits. In [9], hand abstraction is used to verify timed synchronous domino circuits from [1]. Although hand abstraction is extremely effective in verifying these circuits, it requires an expert user and necessitates other methods to validate the abstracted circuit as a reliable model of the actual circuit.

This paper presents an automatic abstraction method that combines the ideas of compositional reasoning with abstraction to avoid additional proof obligations in the verification problem. This paper applies the approach to *timed Petri-net* models of timed circuits to demonstrate its effectiveness. We examine timed circuits because of their inherent modular structure which is ideal for modular verification. The automatic abstraction method verifies each circuit component individually by reducing model complexity through semantics-preserving transformations applied to the connecting components. The transformations remove model details in the connecting components that do not affect the correctness of the component of interest. We prove in this approach that under certain constraints, if each component is verified correct using the semantics-preserving transformations in the connecting fabric, then the complete system is also correct. To show the effectiveness of this approach, this paper presents case studies on a FIFO from SUN Microsystems, the STARI communication circuit, and the RAPPID instruction length decoder. These studies suggest that taking advantage of the structural information in this way reduces the verification cost of timed circuits. It also needs to be pointed out that our method is not constrained to a particular state space search algorithm since it modifies the specification before analysis. In other words, our abstraction technique is complimentary to methods that utilize BDDs or partial orders.

This paper is organized as follows. The next two sections introduce the definitions and semantics of timed Petri-nets. Section IV gives a brief overview of trace structures and introduces basic operations in trace theory that are necessary to prove our modular verification theory. Section V introduces safe abstraction and safe transformations, and gives the conditions that must be satisfied when removing information from a model without reducing the essential properties. Section VI describes several safe transformations to remove the irrelevant information from a model. These transformations are proved to be safe based on the conditions given in Section V. Section VII develops the modular verification theory and proves its correctness. Section VIII gives results of applying our approach to several case studies. This paper concludes with a discussion of the future research that can improve this approach.

II. Timed Petri-Nets: Definitions

Our method uses timed Petri nets [30] to model timed circuits. This section introduces basic definitions of timed Petri nets. The next section presents their semantics.

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.

Definition 2.1: A W-labeled one-safe timed Petri net (TPN) is a directed bipartite digraph described by the tuple $N = (T, P, F, M_0, \mathrm{l}, \mathrm{u}, C, L)$ 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;
- $\mathrm{l} : P \rightarrow \mathbb{Q}^+$ is the lower bound function;
- $\mathrm{u} : P \rightarrow \mathbb{Q}^+ \cup \{\infty\}$ is the upper bound function;
- $C \subseteq P$ is the set of constraint places;
- $L : T \rightarrow (W \times \{+, -\}) \cup \{\$\}$ is the labeling function.

The first four members of the eight tuple define the structure and initial marking of the Petri net. The functions l and u define a lower and upper bound for when tokens can be consumed by transitions. C is used to specify safety and bounded response time properties that must hold in the circuit. Constraint places define ordering requirements and bounded response properties between two arbitrary transitions in a TPN. An error occurs when a transition can fire according to the marking and age of tokens in its ordinary places in P while violating the additional constraints implied by the places in C. The function L maps transitions in a TPN to transitions on wires in W. The symbol \$ denotes the abstracted transitions. These do not affect the state of any wires in W and are place holders for timing information from abstracted signals.

The STARI circuit [31] is used to illustrate our modeling approach. The STARI circuit enables communication between two circuits that are operating at the same clock frequency but are out-of-phase due to clock skew. Clock skew can make it appear that one of the circuits operates faster than the other. The STARI protocol puts more data into a FIFO when the transmitter works faster than the receiver and supplies data from a FIFO to the receiver when the receiver works faster. The STARI circuit is a common timed circuit benchmark, since its correctness depends on timing assumptions. Fig. 1 shows the block diagram of a STARI circuit with two stages.

The functionality of the STARI circuit is described as follows. At the beginning of each clock period, one data is inserted into the FIFO by the transmitter (TX) by setting either *x0.t* or *x0.f* high. At the same time, one data is removed by the receiver

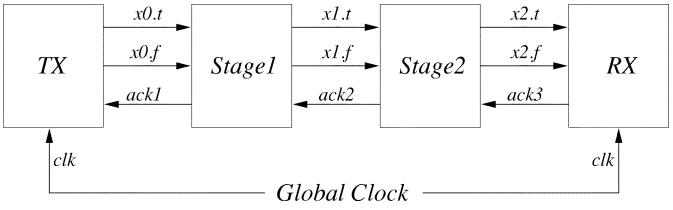


Fig. 1. Block diagram for a two-stage STARI circuit.

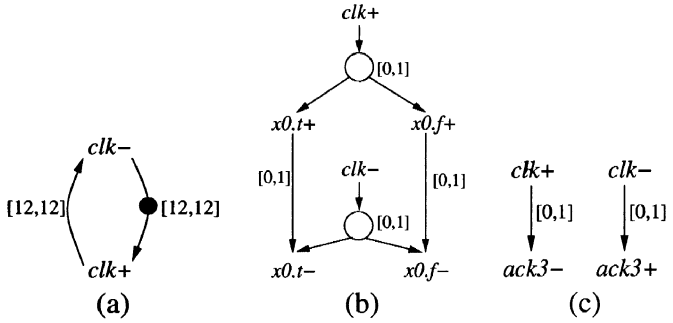


Fig. 2. (a) TPN for the global clock module. (b) The TPN for the TX module. (c) The TPN for the RX module.

(RX) by setting *ack3* low. Data is allowed to propagate asynchronously down the FIFO queue. When *clk* goes low, TX removes the input data by resetting *x0.t* or *x0.f* and RX removes the acknowledgment by setting *ack3* high.

The STARI circuit is modeled in our approach through a collection of TPNs. The TPN for the global clock from the STARI circuit is shown in Fig. 2(a). The labeled transitions from L are shown instead of the members of T to clarify the figures. Note that the places are not shown either. Although places do exist on every edge, they are omitted whenever possible to simplify the figures. The model in Fig. 2(a) toggles the clock with a fixed period of 12 time units. The TPN model for TX is shown in Fig. 2(b). When TX sees the transition *clk+*, it nondeterministically produces either *x0.t+* or *x0.f+*. This is the dual-rail encoded data that is sent to Stage1. The delay interval [0, 1] is used to model the clock skew. After transmitting the data, TX waits for a *clk–*. It then fires either the *x0.t–* or *x0.f–* transition depending on what it transmitted. It fires this transition in the [0, 1] delay interval. The TPN for RX is shown in Fig. 2(c). RX waits for *clk+* and then lowers *ack3* to indicate that it has received the data. After *clk–*, it raises *ack3* to request new data. Again, the delay interval [0, 1] is used to model clock skew.

The TPN for Stage1 of the STARI circuit is presented in Fig. 3(a). The place with dotted arcs is a constraint place and can be ignored for the moment. This stage does not currently have any data, and its *ack1* output is high. It waits for either *x0.t+* or *x0.f+* to indicate valid data and *ack2+* to indicate that Stage2 is empty and ready to receive data. It then fires *x1.t+* or *x1.f+* to pass the data to Stage2. After firing one of these two transitions, it fires *ack1–* to indicate to TX that it has successfully received the data. At this point, it waits for either *x0.t–* or *x0.f–* and *ack2–* from Stage2 to indicate that it has accepted the data. It then resets its data output by firing either *x1.t–* or *x1.f–* depending on the data it transmitted. After firing one of these transitions, it fires *ack1+* to request new data. The TPN model

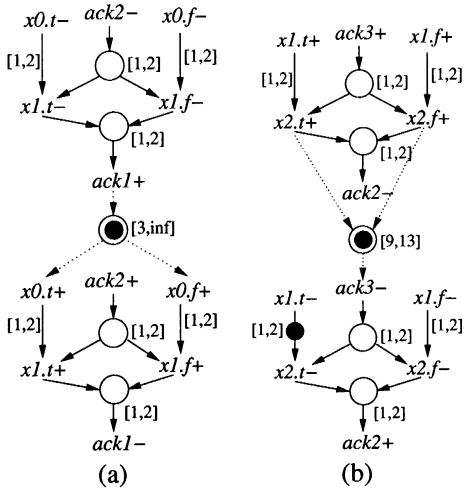


Fig. 3. (a) TPN for Stage1. (b) TPN for Stage2.

for Stage2, shown in Fig. 3(b), is similar to Stage1. The only difference is that Stage2 starts in a different state because it is initialized with a data item.

There are two safety properties in STARI that need to be verified. First, each data output by TX must be inserted into the FIFO before TX sends another data and, second, a new data must be output by the FIFO before each acknowledgment from the receiver [27]. Such properties are checked by places in C. These are often derived from the circuit's specification. Connecting two transitions with a constraint place creates an ordering and bounded response time property that must hold between the two transitions in any run of the system; thus, a place in C is different from a place in P in that it does not affect the firing of a transition. Arcs from constraint places are noncausal. Only places in P are considered when firing a transition. Once a transition is selected to fire, the places in C are checked to make sure that firing that transition does not violate ordering and timing requirements defined in the specification. In a TPN, constraint places are indicated with dotted edges into and out of the places. To model the first property, a constraint place is added to the TPN for Stage1 in Fig. 3(a). A constraint violation occurs whenever TX fires *x0.t+* or *x0.f+* before Stage1 fires *ack1+*. A constraint violation also occurs whenever TX fires *x0.t+* or *x0.f+* after *ack1+*, but the age of the constraint place in the marking is less than 3 (i.e., one of the transitions fired too early). The second property is checked in a similar way in Stage2, as shown in Fig. 3(b), except it requires a bounded response time by *ack3–*. The firing of either *x2.t+* or *x2.f+* can add the constraint place to the marking. Its age in the marking must not fall outside the [9], [13] delay bound before *ack3–* fires in a correct STARI implementation. In order to satisfy this property, however, the FIFO must be initialized to be half full [31]. This is why Stage2 initially contains a data item.

A timed-circuit specification is often a collection of TPNs, as illustrated in the STARI example. Each TPN defines the behavior of a module. The *parallel composition* of a collection of TPN's $N_0 \parallel N_1 \parallel N_2 \parallel \cdots \parallel N_n$ is the single TPN N that is the union over the constituent members of each N_i; thus, $P = \bigcup_{i=0}^{n} P_i$, $T = \bigcup_{i=0}^{n} T_i$, $F = \bigcup_{i=0}^{n} F_i$, etc. Fig. 4 shows

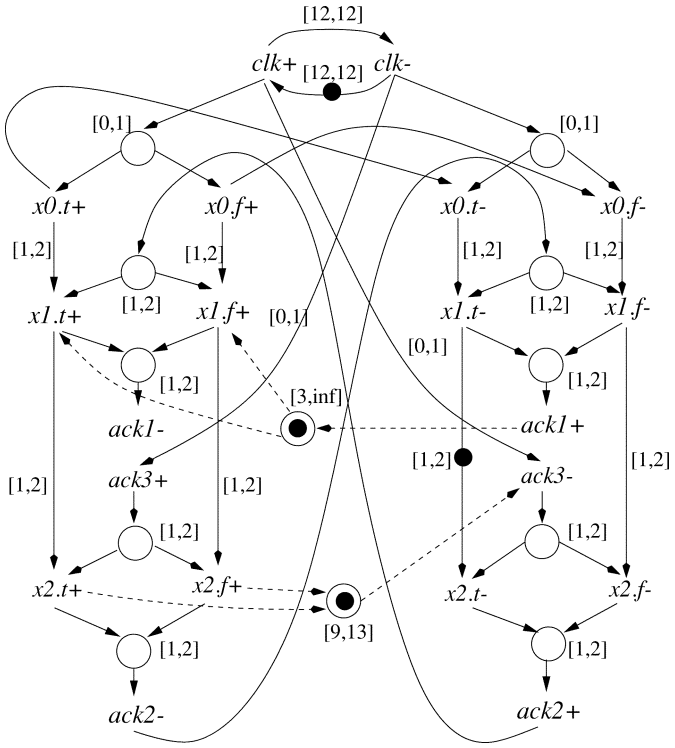


Fig. 4. TPN for the entire two stage STARI circuit.

the parallel composition of the global clock, TX, RX, and the FIFO stages shown in Figs. 2 and 3.

III. Timed Petri-Nets : Semantics

The states of Petri nets are associated with their markings, M, which is the set of places that hold tokens. Our method assumes that correct nets are one-safe so places can only hold a single token.[1] 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., $P(t) = \bullet t - C$. The *postset* of a transition is the set of places that the transition feeds into. For a transition $t \in T$, this is defined as $t\bullet = \{p \in P \mid (t,p) \in F\}$. A transition is *enabled* in a state if the members of its place-set form a subset of the places in the marking of the state.

Definition 3.1: The transition t is **enabled** in a marking M if $P(t) \subseteq M$. The set of transitions enabled in M is denoted by $X(M)$.

The state of a TPN contains, beside the marking, timing information associated with each place, i.e., its age, the time since its last addition to the marking. The state of a TPN is a pair (M, D) where M is the current marking and $D : P \to \mathbb{R}^+$ is a clock assignment function assigning positive reals 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, for the system is defined such that the clock for every place is zero. The initial state of the TPN $N = (P, T, F, M_0, \mathrm{l}, \mathrm{u})$ is given by 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 in l and u. A transition is *time-enabled* if two conditions are met: first, the clock for each place in its place-set is above its lower timing bound; and second, there exists a clock for a place in its place-set that is below its upper timing bound. These two conditions are formalized in Definition 3.2.

Definition 3.2: An enabled transition, t, is **time-enabled** in the state (M, D) if for all $p \in P(t)$, $D(p) \geq \mathrm{l}(p)$; and there exists a place $p' \in P(t)$ such that $D(p') \leq \mathrm{u}(p')$.

These semantics allow some places to exceed their upper bound while requiring all places to meet their lower bound before a transition fires. Causality is not strictly determined by the order of arrival of the tokens in the preset of a transition unless all places in the preset have identical bounds. Rather, it is determined by the age of the tokens in the preset and their respective bounds. Although a token in a preset does not arrive last, if it has a sufficient delay bound it can determine the latest firing time of the transition. 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 the TPN can change not only by firing transitions, but also 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 cannot advance indefinitely in a state but only by an amount that does not disable a time-enabled transition.

Definition 3.3: The **maximum delay** $d \in \mathbb{R}^+$ that can advance at a state (M, D) is

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

After advancing time by the maximum delay, a transition either remains nontimed-enabled, becomes time-enabled, or is already time-enabled and remains so.

Consider again the TPN shown in Fig. 2(a). In the initial marking, the only enabled transition is *clk+* which fires after time is advanced by 12 time units. This enables *clk–* which fires after another 12 time units have advanced. Consider now the TPN shown in Fig. 2(b). Since *clk+* has no places in its preset, it can fire at any time. After *clk+* fires, either *x0.t+* or *x0.f+* can fire or time can advance by up to 1 time unit. The TPN's in this figure describe only part of the behavior of the STARI circuit.

Let us now consider the composite TPN shown in Fig. 4. In the initial marking, *clk+* is the only enabled transition which fires after time is advanced by 12 units. The firing of *clk+* results in a marking where there are four enabled transitions *clk–*, *x0.t+*, *x0.f+*, and *ack3–*. Since the lower bound of the timing constraint on *x0.t+*, *x0.f+*, and *ack3–* is zero, these transitions can fire immediately, and time cannot advance more than one time unit before one of them is forced to fire. Note that since *clk–* must wait at least 12 time units, it cannot fire first. Assume

[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.

that *x0.t+* fires first. This firing disables *x0.f+* as it consumes the token in their shared place.

Let N be a TPN. A *trace* of N is a sequence of transition-time pairs, (t, τ), such that the transitions and their labels are taken from N. The time, τ, is a time stamp reference from the beginning of the trace. The amount of time elapsed between any two consecutive entries, $i-1$ and i, in the trace can be computed from the values of τ_{i-1} and τ_i. In the case of the first entry in the trace, $\tau_0 = 0$. A trace is *valid* in N if starting from the initial state pair of N, advancing time by the specified delay in the trace and then firing the transition leads to a state pair where the next time advance and transition firing in the trace can occur. This is expressed in Definition 3.4.

Definition 3.4: The trace $((t_1, \tau_1), (t_2, \tau_2), \ldots, (t_n, \tau_n))$ is a **valid trace** if there exists a sequence of states $S = (s_0, s_1, s_2, \ldots, s_{n-1}, s_n)$ such that for $1 \leq i \leq n$ each $s_i = (M_i, D_i)$ and $d_i = \tau_i - \tau_{i-1}$,

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 traces for a TPN N starting from an initial state (M_0, D_0) is denoted by $\mathcal{P}(N)$.

A set $\mathcal{P}$ of traces is *prefix-closed* if $x \in \mathcal{P}$ implies that every prefix of x is in $\mathcal{P}$. The set $\mathcal{P}(N)$ of valid traces is prefix-closed by definition. A trace is *monotonic* if time can only advance in the forward direction. A valid trace is also monotonic by definition.

The set of possible traces in a TPN can be divided into those that are considered successes and those that are considered failures. There are three types of failure modeled in the TPN: *safety*; *constraint*; and *complement*. The *state-set* of a trace $S(x)$, is the sequence of states created by the trace starting from the initial state of the TPN. A valid trace of a TPN is a *safety failure* if in firing the trace the marking update tries to add to the new marking a place that already exists in the current marking. This is formalized in Definition 3.5.

Definition 3.5: The valid trace $x = ((t_1, \tau_1), \ldots, (t_n, \tau_n))$ generates a **safety failure** if given its state sequence $S(x) = (s_0, \ldots, s_n)$ there exists a state $s_{i-1} = (M_{i-1}, D_{i-1})$ and transition-time pair (t_i, τ_i) where $(M_{i-1} - {\bullet}t_i) \cap t_i{\bullet} \neq \emptyset$.

Consider again the TPN shown in Fig. 4. If timing is ignored, then from the initial marking *clk+* could fire followed by *clk–* and *clk+* again. In computing the new marking from firing *clk+*, a place is added to the marking that already exists in the marking. This is a safety failure. The place is part of the preset for transitions *x0.t+* and *x0.f+*, and it persists in the marking until one of these two transitions fire. Any trace that does not separate consecutive *clk+* transitions with intervening *x0.t+* or *x0.f+* transitions is a safety failure.

A valid trace is a *constraint failure* if it contains a transition or time progress that could not have occurred if constraints places are taken into account in the definition of enabledness.

Definition 3.6: The valid trace $x = ((t_1, \tau_1), , \ldots, (t_n, \tau_n))$ generates a **constraint failure** if given its corresponding state sequence $S(x) = (s_0, , \ldots, s_n)$, there exists a state $s_{i-1} = (M_{i-1}, D_{i-1})$ and delay, $d_i = \tau_i - \tau_{i-1}$, such that one of the three following conditions hold:

1) there exists a constraint place $p \in {\bullet}t_i \cap C$ that satisfies $(p \notin M_{i-1}) \vee (D(p) + d_i < \mathrm{l}(p))$; or
2) there exists a constraint place $p \in C \cap M_{i-1}$ that satisfies $D(p) + d_i > \mathrm{u}(p)$.
3) $X(M_i) = \emptyset \wedge c \in M_i$

There are three failure conditions corresponding to constraint places. The first type of failure occurs when 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. The second type occurs when a token stays in a constraint place beyond its upper bound. The third type occurs when a trace deadlocks with constraint places still marked.

To further illustrate the role of constraint places, recall that there are two properties in STARI that need to be verified: first, each data output by TX must be inserted into the FIFO before TX sends another data and, second, a new data must be output by the FIFO before each acknowledgment from RX. The bounds on the two constraint places refine these properties with the following information: first, TX must separate data by at least three time units and, second, a new data must be output by the FIFO at least nine and no more than 13 time units before the acknowledgment from RX. The constraints enable the specification of the bounded response time properties necessary to interface the STARI circuit with the two skewed clock domains without introducing race conditions or setup and hold time violations. The constraint is violated, however, if the bound is changed to [10] and [12]. In this case, *ack3–* can fire as early as nine time units after *x2.t+* or *x2.f+* by the observation that its constraint place can be marked as late as three time units after *clk–*, and *clk+* is forced to fire in another nine time units. Transition *ack3–* can also fire as late as 13 time units after *x2.t+* or *x2.f+* in the initial marking by the observation that *clk+* fires 12 time units after *clk–*, and *ack3–* can fire one time unit later. Constraints can be systematically derived from the functional specification of the system and iteratively checked as the model is refined with implementation detail.

A valid trace is a *complement failure* if there exists in the trace a transition-time pair that fires a rising or falling transition on a signal that is already in a high or low Boolean state, respectively. This is formalized in Definition 3.7.

Definition 3.7: The valid trace $x = ((t_1, \tau_1), \ldots, (t_n, \tau_n))$ generates a **complement failure** if there exists two pairs (t_i, τ_i) and (t_k, τ_k) such the following three conditions hold:

1) $i < k$;
2) $L(t_i) = L(t_k) \wedge L(t_i) \neq \$$;
3) for all transition-time pairs (t_j, τ_j) such that $i < j < k$, $t_j \neq t_k$, $(L(t_i) = s+ \implies L(t_j) \neq s-)$, and $(L(t_i) = s- \implies L(t_j) \neq s+)$.

The first condition forces t_i to occur before t_k in the trace. The second condition forces t_i to be the same transition as t_k on the same wire in W, and the last condition forces that t_k is the first instance of a transition identical to t_i without an intervening opposite transition; thus, a complement failure trace fires the same transition at least two times in a row without first toggling

the signal to its opposite Boolean state. Consider the TPN shown in Fig. 2(b). Since there are no places in the preset of *clk+*, it can fire at any time. Therefore, a possible trace for this TPN is ((*clk+*,1),(*clk+*,2)). This trace is, however, a failure since *clk+* has occurred two times in a row without an intervening *clk-*.

Let N again be a TPN and $\mathcal{X} \subseteq \mathcal{P}(N)$ be a set of valid traces in N. The function $\mathbf{fail}(\mathcal{X})$ returns the traces in $\mathcal{X}$ that are either safety, constraint, or complement failures. The goal is to show that $\mathbf{fail}(\mathcal{P}(N)) = \emptyset$ through modular verification using abstraction because the entire reachable state space of N is too large to enumerate for a real design. The fail function must never hide failure traces for this approach to work; thus, any definition of **fail** must satisfy the following property:

$$\mathbf{fail}(\mathcal{X}) \subseteq \mathbf{fail}(\mathcal{X}') \quad \text{if } \mathcal{X} \subseteq \mathcal{X}' \tag{1}$$

where $\mathcal{X}$ and $\mathcal{X}'$ are two subsets of valid traces in $\mathcal{P}(N)$. It can be shown that a definition of **fail** that includes safety, constraint, and complement failures satisfies this condition.

IV. Basic Trace Theory

Trace-based models of concurrent processes were proposed by Hoare [32], Milner [33], and others. *Trace theory* has since been developed and applied to the verification of both speed-independent [3], [34] and timed circuits [11], [35]. Since the dynamic behavior of a system can be described using the set of possible timed traces it produces, it can also be modeled using a *timed trace structure*. A timed trace structure $\mathcal{T}$ is a four-tuple $\langle \mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{F} \rangle$ where $\mathcal{I}$ is the set of input transitions, $\mathcal{O}$ is the set of output transitions, $\mathcal{S}$ is the set of successful traces, and $\mathcal{F}$ is the set of failure traces. This paper requires that the sets of input and output transitions be disjoint (i.e., $\mathcal{I} \cap \mathcal{O} = \emptyset$). $\mathcal{A} = \mathcal{I} \cup \mathcal{O}$ is the *alphabet* of $\mathcal{T}$. The set of all possible timed traces is denoted by $\mathcal{P}$, and $\mathcal{P} = \mathcal{S} \cup \mathcal{F}$.

The function $\mathcal{T}(N)$ takes a TPN N and returns its trace structure. This function creates a trace structure of the form

$$\begin{aligned} \mathcal{I} &= \{t \in T \mid P(t) = \emptyset\} \\ \mathcal{O} &= T - \mathcal{I} \\ \mathcal{S} &= \mathcal{P}(N) - \mathcal{F} \\ \mathcal{F} &= \mathbf{fail}\big(\mathcal{P}(N)\big). \end{aligned}$$

The input transitions are those that have been left unconstrained (i.e., the only places in their preset are constraint places) while the output transitions are all other transitions. Note that by construction, a trace structure is *receptive* (i.e., if $x \in \mathcal{P}$, then xe where e is an input transition event is also in $\mathcal{P}$). In other words, a circuit cannot prevent the environment from causing an input transition. The failure traces are those valid traces in $\mathcal{P}(N)$ that are returned by the function **fail** defined in the last section, while the success traces are all the other possible traces. Note that by construction $\mathcal{S} \cap \mathcal{F} = \emptyset$. A trace structure is said to be *failure-free* if $\mathcal{F} = \emptyset$.

The rest of this section gives a brief introduction to the basics of trace theory, including basic operations on trace structures and lemmas that are necessary to prove the lemma and theorems in the following sections. More details can be found in [3]. Although all the definitions and proofs are proposed for untimed traces, they are naturally extended to timed traces in which the timing portion of the trace is ignored.

The delete function $\mathbf{del}(\mathcal{D})(x)$ removes all events of a trace $x = e_1 e_2 \ldots$ whose transitions are in a set $\mathcal{D}$. 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 $\mathbf{del}(\mathcal{D})(x) = \{\epsilon\}$. It is extended naturally to sets of traces. The inverse delete function, $\mathbf{del}^{-1}(\mathcal{D})(\mathcal{X})$, takes a set of transitions, $\mathcal{D}$, and a set of traces, $\mathcal{X}$, and returns the set of traces which would be in $\mathcal{X}$ if all events with transitions in $\mathcal{D}$ are deleted (i.e., $\mathbf{del}^{-1}(\mathcal{D})(\mathcal{X}) = \{x' \mid \mathbf{del}(\mathcal{D})(x') \in \mathcal{X}\}$). Intuitively, if x is a trace not containing symbols from $\mathcal{D}$, $\mathbf{del}^{-1}(\mathcal{D})(x)$ is the set of all traces that can be generated by inserting events in $\mathcal{D}$ at any time into x. Some useful properties of these two functions are [3], [36]

$$\mathbf{del}(\mathcal{D})(\mathcal{X}) = \emptyset \Leftrightarrow \mathcal{X} = \emptyset \tag{2}$$

$$\mathbf{del}(\mathcal{D})(\mathbf{del}^{-1}(\mathcal{D}')(\mathcal{X})) = \mathbf{del}^{-1}(\mathcal{D}')(\mathbf{del}(\mathcal{D})(\mathcal{X})) \quad \text{when } \mathcal{D} \cap \mathcal{D}' = \emptyset \tag{3}$$

$$\mathbf{del}(\mathcal{D})(\mathbf{del}^{-1}(\mathcal{D})(\mathcal{X})) = \mathcal{X} \tag{4}$$

$$\mathbf{del}(\mathcal{D})(\mathcal{X} \cap \mathcal{X}') \subseteq \mathbf{del}(\mathcal{D})(\mathcal{X}) \cap \mathbf{del}(\mathcal{D})(\mathcal{X}'). \tag{5}$$

A useful operation is **hide** that is used to make a set of transitions, $\mathcal{D} \subseteq \mathcal{O}$, *internal* to the circuit. Given a trace structure $\mathcal{T}$, $\mathbf{hide}(\mathcal{D})(\mathcal{T})$ is defined as follow:

$$\mathbf{hide}(\mathcal{D})(\mathcal{T}) = \langle \mathcal{I},\ \mathcal{O} - \mathcal{D},\ \mathbf{del}(\mathcal{D})(\mathcal{P}) - \mathcal{F},\ \mathcal{F} \rangle$$

where $\mathcal{F} = \mathbf{fail}(\mathbf{del}(\mathcal{D})(\mathcal{P}))$. Note that our definition of **hide** is different from that in [3] in that errors on transitions in $\mathcal{D}$ can be hidden in our definition. **hide** can be used to remove the internal details of one module without affecting the correctness of the module in which we are interested.

Composition ($\|$) combines two trace structures into a single trace structure. Composition of two trace structures $\mathcal{T} = \langle \mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{F} \rangle$ and $\mathcal{T}' = \langle \mathcal{I}', \mathcal{O}', \mathcal{S}', \mathcal{F}' \rangle$ is defined when $\mathcal{O} \cap \mathcal{O}' = \emptyset$. To compose two trace structures, the alphabets of both trace structures must first be made the same by adding new inputs as necessary to each structure. Inverse delete is extended to trace structures for this step as follows:

$$\mathbf{del}^{-1}(\mathcal{D})(\mathcal{T}) = \langle \mathcal{I} \cup \mathcal{D},\ \mathcal{O},\ \mathbf{del}^{-1}(\mathcal{D})(\mathcal{S}),\ \mathbf{del}^{-1}(\mathcal{D})(\mathcal{F}) \rangle.$$

This is defined only when $\mathcal{D} \cap \mathcal{A} = \emptyset$. After the two alphabets of the two structures are made to match, the trace structures are intersected to find the traces that are consistent with the two structures. The intersection of these two trace structures is defined as follows:

$$\mathcal{T} \cap \mathcal{T}' = \langle \mathcal{I} \cap \mathcal{I}',\ \mathcal{O} \cup \mathcal{O}',\ \mathcal{S} \cap \mathcal{S}',\ (\mathcal{F} \cap \mathcal{P}') \cup (\mathcal{P} \cap \mathcal{F}') \rangle.$$

This is defined when $\mathcal{A} = \mathcal{A}'$ and $\mathcal{O} \cap \mathcal{O}' = \emptyset$. A success trace in the composite must be a success trace in both components. A failure trace in the composite is a possible trace that is a failure

trace in either component. The possible traces for the composite are $\mathcal{P} \cap \mathcal{P}'$. Composition can now be defined

$$\mathcal{T} \parallel \mathcal{T}' = \mathbf{del}^{-1}(\mathcal{A}' - \mathcal{A})(\mathcal{T}) \cap \mathbf{del}^{-1}(\mathcal{A} - \mathcal{A}')(\mathcal{T}').$$

An important property used in trace theory is *conformance*. When a trace structure $\mathcal{T}$ conforms to another trace structure $\mathcal{T}'$, it is safe to substitute the circuit modeled by $\mathcal{T}$ whenever the circuit modeled by $\mathcal{T}'$ is required. Conformance is defined as follows:

Definition 4.1: Given two trace structures, $\mathcal{T}$ and $\mathcal{T}'$, we say $\mathcal{T}$ *conforms* to $\mathcal{T}'$ (denoted $\mathcal{T} \preceq \mathcal{T}'$) if $\mathcal{I} = \mathcal{I}', \mathcal{O} = \mathcal{O}'$, and for *all* environments $\mathcal{E}$, if $\mathcal{E} \parallel \mathcal{T}'$ is failure-free, so is $\mathcal{E} \parallel \mathcal{T}$.

Intuitively, if a system using a circuit modeled by $\mathcal{T}'$ cannot fail, neither can a system using a circuit modeled by $\mathcal{T}$.

Lemma 4.1 below gives a simple sufficient condition to determine conformance between two trace structures. The condition $\mathcal{F} \subseteq \mathcal{F}'$ assures that if the environment does not cause a failure in $\mathcal{T}'$, it does not cause a failure in $\mathcal{T}$. The condition $\mathcal{P} \subseteq \mathcal{P}'$ assures that if $\mathcal{T}'$ does not cause a failure in the environment, $\mathcal{T}$ does not cause one, either. Lemma 4.2 shows that if $\mathcal{T}$ conforms to $\mathcal{T}'$, this conformance is maintained in any environment. Proofs of these lemmas can be found in [3].

Lemma 4.1: $\mathcal{T} \preceq \mathcal{T}'$ if $\mathcal{I} = \mathcal{I}', \mathcal{O} = \mathcal{O}', \mathcal{F} \subseteq \mathcal{F}'$, and $\mathcal{P} \subseteq \mathcal{P}'$.

Lemma 4.2: If $\mathcal{T} \preceq \mathcal{T}'$ and $\mathcal{T}''$ is any trace structure, then $\mathcal{T} \parallel \mathcal{T}'' \preceq \mathcal{T}' \parallel \mathcal{T}''$.

V. Automatic Abstraction

In general, a large and complex design is organized as a number of components, each of which has a well-defined interface. In a system consisting of multiple components, each component either connects to other components, the environment, or both. Since the complexity of each component is often much less than the whole system, it is desirable to verify each component individually, and integrate the results for all components when available to form the solution for the whole system. If a component is chosen for verification, the rest of the components and the system environment together form the environment in which the selected component operates. To verify a component, only the interface behavior of the environment is important to the component. Therefore, if the internal behavior of the environment is abstracted away while preserving its interface behavior, the environment can be simplified to reduce the complexity of verification.

To apply abstraction to TPNs, first, a component is selected, and then the TPNs for all other modules are composed together to form the environment for the selected component. For the STARI example, if Stage1 is selected, then the TPN for its environment is created by composing the TPN's for the CLK, TX, RX, and Stage2 as shown in Fig. 5.

In the second step, all internal signals of the environment relative to a chosen component are identified and transitions on them are placed in the set $\mathcal{D}$. The transitions in $\mathcal{D}$ are called the *abstracted transitions*. Our method also removes each constraint place that has abstracted signals in its preset and postset (i.e., $P = P - C'$ and $C = C - C'$, where $C' = C \cap \{p \in P \mid (\bullet p \cup p\bullet) \subseteq \mathcal{D}\}$), as well as each reference to it in the flow relation (i.e., $F = F - \{(t,p) \text{ or } (p,t) \mid p \in C'\}$). Our method also does not consider the abstracted transitions when determining if there is a complement failure. They are, however, still considered for safety failures.

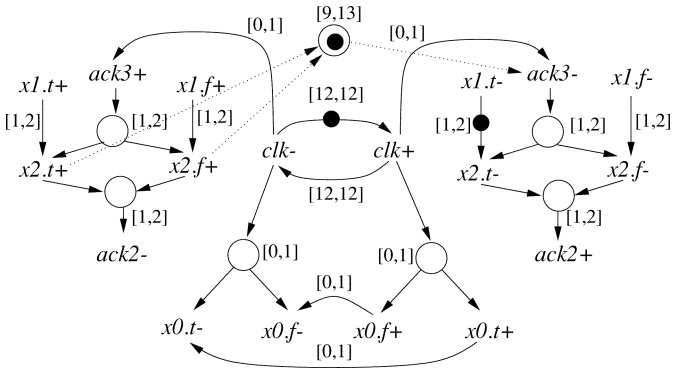


Fig. 5. Composition of TPN's for CLK, TX, RX, and Stage2 from the STARI example to form the environment for Stage1.

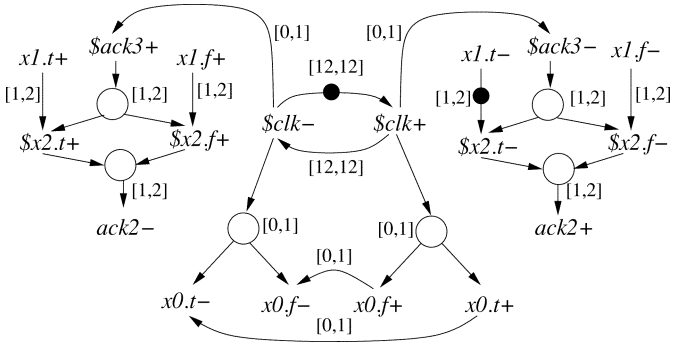


Fig. 6. Environment for Stage1 after abstracting internal transitions.

Consider the environment for Stage1 shown in Fig. 5. If we are verifying only the first stage of the two stage FIFO, then the signals *ack3*, *x2.t*, *x2.f*, and *clk* are not on the interface of the first FIFO stage and should be abstracted. Transitions on these signals are marked as shown in Fig. 6, where the name of each transition has been preceded by a '$'. The constraint place between the transitions *x2.t+* and *x2.f+* and the transition *ack3-* is also removed.

Simply removing these transitions from the definition of **fail** does not substantially reduce the complexity of analysis since the transitions still occur and cause changes in the marking. Therefore, the third step to the abstraction process is to remove these abstracted transitions and the related places whenever it can be done safely. In other words, the TPN resulting from these *transformations* must produce a superset of the timed traces produced by the original TPN when only the interface behavior is considered. Such transformations are defined to be *safe*. The notions of safe abstraction and safe transformation are formally defined as follows:

Definition 5.1 (Safe Abstraction): A TPN N' is a safe abstraction of a TPN N if $del(\mathcal{D})(\mathcal{P}(N)) \subseteq \mathcal{P}(N')$ where $\mathcal{D} = \mathcal{O} - \mathcal{O}'$.

Definition 5.2 (Safe Transformation): A safe transformation is a transformation that when applied to a TPN N generates a new TPN N' that is a safe abstraction of N.

Determining if a transformation is safe can be broken down into two steps. First, a safe transformation must not reduce the specified untimed behavior of the TPN. Second, the timing information carried by the TPN must be preserved in a conservative fashion. We define a function $\mathbf{abs}(\mathcal{D})(N)$ that takes a set of signal transitions $\mathcal{D}$ and a TPN N and removes transitions in $\mathcal{D}$ from N using safe transformations, when possible.

Since safe transformations only add traces and adding traces does not remove failures [see (1)], we know that this approach never results in a *false positive* answer (i.e., a system verified failure-free with the abstracted environment while verification would detect failure traces in the system with the flat environment). However, verifying a system with a transformed environment may result in a *false negative* answer. A false negative is when a failure trace is found while verifying a system with the abstracted environment, but no failure traces are possible if the system is verified with the original environment. When a failure trace is reported, this trace is examined to see if it is a possible trace for the flat system. If this trace is not a possible trace, then this may be a false negative (note that it may still be a true failure but a false failure trace). If the failure trace is found not to be a possible trace, it is necessary to determine which transformation causes the false negative and rerun the abstraction and verification process without this transformation. This is accomplished by incrementally backing off transformations that are known to more likely create a false negative. If false negatives happen rarely, transformations can be used to substantially reduce the complexity of the analysis.

Calculating the interface behavior of a component's environment can be conducted by state space exploration to generate all possible timed traces and then applying the function $\mathbf{del}$ to remove all internal signal transitions from the possible timed traces. However, state space exploration is an exponential problem, which is computationally infeasible for large systems. Instead, internal signal transitions should be removed from the TPN representing the component's environment using safe transformations before state space exploration starts. Since only safe transformations are utilized, the possible timed traces produced by the abstracted TPN include the specified interface behavior. The following lemma proves that the unabstracted version of the component's environment conforms to the abstracted version on the same set of interface signal transitions. This conformance is used to develop a sound methodology for modular verification discussed in Section VII.

Lemma 5.1: Let N be a TPN, $\mathcal{O}$ be the set of output transitions in the trace structure $\mathcal{T}(N)$, and $\mathcal{D} \subseteq \mathcal{O}$ be the set of internal transitions. If the function $\mathbf{abs}(\mathcal{D})(N)$ uses only safe transformations, then $\mathbf{hide}(\mathcal{D})(\mathcal{T}(N)) \preceq \mathcal{T}(\mathbf{abs}(\mathcal{D})(N))$.

Proof: Let $\mathcal{P}$ and $\mathcal{P}'$ be the possible trace sets of $\mathbf{hide}(\mathcal{D})(\mathcal{T}(N))$ and $\mathcal{T}(\mathbf{abs}(\mathcal{D})(N))$, respectively. From the definition of safe transformations, we have $\mathcal{P} \subseteq \mathcal{P}'$. From (1), we have $\mathbf{fail}(\mathcal{P}) \subseteq \mathbf{fail}(\mathcal{P}')$. Therefore, from Lemma 4.1, we have $\mathbf{hide}(\mathcal{D})(\mathcal{T}(N)) \preceq \mathcal{T}(\mathbf{abs}(\mathcal{D})(N))$. ■

VI. Safe Transformations

This section describes the safe transformations used to remove abstracted transitions from TPNs. Suzuki and Murata [37], [38] present a method of stepwise refinement of transitions and places into subnets. They show a sufficient condition that such subnets must satisfy which is dependent on the structure and initial marking of the net. Berthelot [39] presents several transformations that depend only on the structure of the net. Murata presents several transformations for marked graphs in [40]–[42]. All these transformations reduce places and transitions in the graph, while preserving liveness and safety properties. These transformations, however, are only applied to untimed Petri nets. We have developed several safe transformations for TPN's. These transformations are safe in that a TPN after these transformations are applied produces a superset of timed traces produced by the original net.

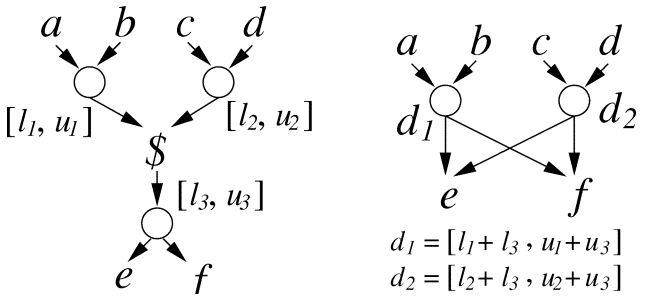


Fig. 7. Safe Transformation 1.

Fig. 7 shows Transformation 1 which is applied when an abstracted transition, \$, has only a single place in its postset. It removes the abstracted transition \$, and the place in the postset of \$. The timing constraint of the place in the postset of \$ is added to those of the places in the preset of \$. Transformation 1 is defined as follows:

Transformation 1: Let \$ be an abstracted transition in a TPN N where $\$\bullet = \{\eta\}$, $\bullet\eta = \{\$\}$, $(\bullet\$)\bullet = \{\$\}$, and either $\eta \notin M_0$ or $\bullet\$ \cap M_0 = \emptyset$, a new TPN N' can be derived from N as follows:

- $T' = T - \{\$\}$;
- $P' = P - \{\eta\}$;
- $F' = (F - R_1 - (\$, \eta)) \cup R_2$ where $R_1 = \{(p,t) \mid (p \in \bullet\$ \wedge t = \$) \vee (p = \eta \wedge t \in \eta\bullet)\}$, and $R_2 = \{(p,t) \mid p \in \bullet\$ \text{ and } t \in \eta\bullet\}$;
- if $\eta \in M_0$ then $M_0' = M_0 - \{\eta\} \cup \bullet\$$;
- $\mathrm{l}'(p) = \mathrm{l}(p) + \mathrm{l}(\eta)$ for $p \in \bullet\$$;
- $\mathrm{u}'(p) = \mathrm{u}(p) + \mathrm{u}(\eta)$ for $p \in \bullet\$$;
- $C' = C$;
- $L' = L$ with \$ removed from the domain.

Lemma 6.1: Transformation 1 is a safe transformation.

Proof: To prove this lemma, we need to show that N and N' produce the same set of untimed traces, and the timing bound from the place η is preserved.

In N and N', the behavior of transitions a, b, c, and d are defined by the environment. To prove that N and N' produce the same set of untimed traces, we need to show that they produce the same set of untimed traces in any environment. This requirement allows these transitions to change arbitrarily. Fig. 8 shows the state diagrams that represent all possible untimed traces produced by N and N' in Fig. 7, respectively. In the state diagrams, s0 and F denote the initial state and failure state of the TPNs.

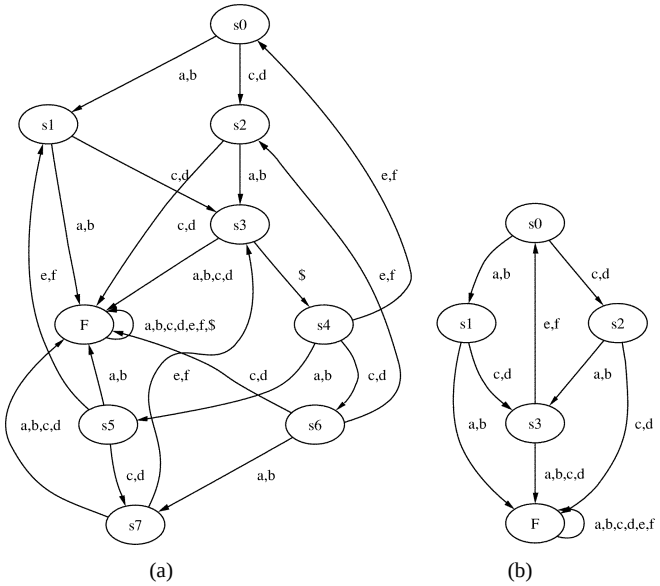


Fig. 8. State diagrams for (a) N and (b) N' in Fig. 7.

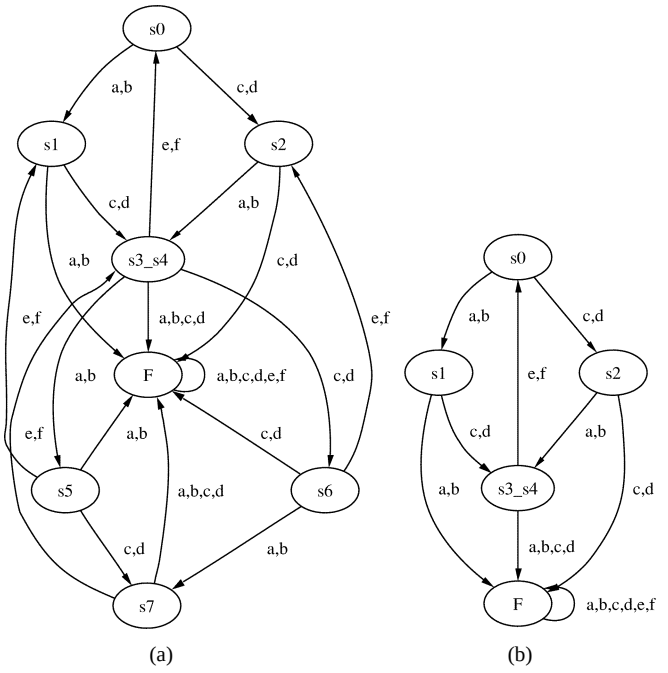


Fig. 9. (a) State diagrams for N after hiding \$, and (b) State diagram for (a) after removing traces which have the prefixes leading to the failure state.

After hiding transition \$, states s3 and s4 in Fig. 8(a) are merged together to become a new state s3_s4 shown in the state diagram in Fig. 9(a). In the state s3_s4, a transition on a, b, c, or d can cause N to fail or succeed nondeterministically. Since a trace that can possibly cause a failure is as bad as a trace that does cause a failure, it is necessary to eliminate all traces from $\mathcal{S}$ that also appear in $\mathcal{F}$. To accomplish this, we delete states s5 and s6 from the state diagram. We can also remove s7, as it is no longer reachable. The new state diagram is shown in Fig. 9(b). By comparing Fig. 9(b) with Fig. 8(b) for N', they are the same if the labels for the states are ignored. Therefore, N and N' produce the same set of possible untimed traces on the transitions in N'.

Consider a timed trace $e_1e_2\ldots$ in which $e_i = (a, \tau_a)$, $e_j = (c, \tau_c)$, $e_k = (\$, \tau_\$)$, and $e_l = (f, \tau_f)$ with $i < k$, $j < k$, and $k < l$. The value of $\tau_\$$ falls in the following range:

$$\max\{\tau_a + l_1, \tau_c + l_2\} \le \tau_\$ \le \max\{\tau_a + u_1, \tau_c + u_2\} \quad (6)$$

The value of τ_f comes from the range

$$\tau_\$ + l_3 \le \tau_f \le \tau_\$ + u_3. \quad (7)$$

Substituting (6) into (7) yields

$$\max\{\tau_a + l_1, \tau_c + l_2\} + l_3 \le \tau_f \le \max\{\tau_a + u_1, \tau_c + u_2\} + u_3. \quad (8)$$

In N', the value of τ_f comes from the range

$$\max\{\tau_a + l_1 + l_3, \tau_c + l_2 + l_3\} \le \tau_f \le \max\{\tau_a + u_1 + u_3, \tau_c + u_2 + u_3\}. \quad (9)$$

This is equivalent to (8), so the ranges of values for τ_f in N and N' are equal. Note that similar results can be obtained for other combinations of events. And since N and N' produce the same set of untimed traces on T', N' produces the same set of timed traces as N on T'. According to the definition of safe transformation, Transformation 1 is safe. ■

Transformation 2 shown in Fig. 10 is applied when the abstracted transition has only a single place in its preset. The following gives the definition of Transformation 2 .

Transformation 2: Let \$ be an abstracted transition in a TPN N where $\bullet\$ = \{\eta\}$, $\eta\bullet = \{\$\}$, $\bullet(\$\bullet) = \{\$\}$, and either $\eta \notin M_0$ or $\$ \bullet \cap M_0 = \emptyset$, a new TPN N' can be derived from N as follows:

- $T' = T - \{\$\}$;
- $P' = P - \{\eta\}$;
- $F' = (F - R_1 - (\eta, \$)) \cup R_2$ where $R_1 = \{(t,p) \mid (t = \$ \wedge p \in \$\bullet) \vee (t \in \bullet\eta \wedge p = \eta)\}$, and $R_2 = \{(t,p) \mid t \in \bullet\eta \text{ and } p \in \$\bullet\}$;
- if $\eta \in M_0$ then $M_0' = M_0 - \{\eta\} \cup \$\bullet$;
- $\mathrm{l}'(p) = \mathrm{l}(p) + \mathrm{l}(\eta)$ for $p \in \$\bullet$
- $\mathrm{u}'(p) = \mathrm{u}(p) + \mathrm{u}(\eta)$ for $p \in \$\bullet$;
- $C' = C$;
- $L' = L$ with \$ removed from the domain.

Lemma 6.2: Transformation 2 is a safe transformation.

Proof: Similar to the proof of Transformation 1, we can show that N and N' in Fig. 10 produce the same set of untimed traces (proof omitted to save space).

Next, we must show that the timing bound from the place η is preserved. Consider a timed trace $e_1e_2\ldots$ in which $e_i = (a, \tau_a)$, $e_j = (\$, \tau_\$)$, $e_k = (d, \tau_d)$, and $e_l = (f, \tau_f)$ with $i < j$, $j < k$, and $j < l$. The value of τ_d falls in the following range:

$$\tau_a + l_1 + l_3 \le \tau_d \le \tau_a + u_1 + u_3. \quad (10)$$

Given the value of τ_d, we can show that τ_f must come from the range

$$\tau_d + l_2 - u_1 \le \tau_f \le \tau_d + u_2 - l_1. \quad (11)$$

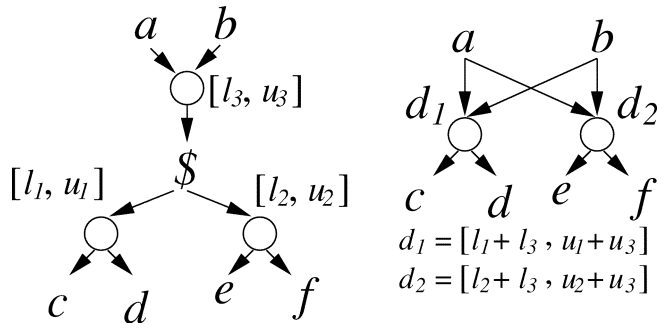


Fig. 10. Safe Transformation 2.

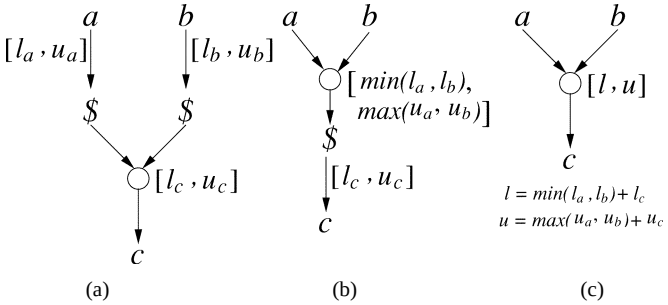


Fig. 11. Safe Transformation 3.

After transformation, the value of τ_d can still be drawn from (10), but the value of τ_f given the value of τ_d must now come from the range

$$\tau_d + (l_2 + l_3) - (u_1 + u_3) \leq \tau_f \leq \tau_d + (u_2 + u_3) - (l_1 + l_3).$$

This can be rewritten as follows:

$$\tau_d + (l_2 - u_1) + (l_3 - u_3) \leq \tau_f \leq \tau_d + (u_2 - l_1) + (u_3 - l_3).$$

Since $l_3 - u_3 \leq 0$ and $u_3 - l_3 \geq 0$, the range of values for τ_f after abstraction is a superset of those before abstraction. Note that similar results can be shown for other sequences of events in the net. Since N and N' produce the same set of untimed traces, N' produces a superset of the timed traces produced by N. According to the definition of safe transformation, Transformation 2 is safe. ■

Although Transformation 1 adds no extra behavior, Transformation 2 may create extra timed traces. For example, in Fig. 10, N' could generate a trace $(a, \tau_a)(d, \tau_a+l_1+l_3)(f, \tau_a+u_2+u_3)$, where τ_a is the time when transition a fires. This trace is impossible in N.

The third transformation involves a merge place and a pair of abstracted transitions, and it is depicted in Fig. 11(a). Transformation 3 is applied when some abstracted transitions have the same postset. This transformation merges the pair of abstracted transitions and the places in their preset into a single abstracted transition and a single place as shown in Fig. 11(b). At this point, Transformation 1 or Transformation 2 can often be applied as shown in Fig. 11(c). Like Transformation 2, this transformation may add additional timing behavior. However, if $l_a = l_b$ and $u_a = u_b$, then it is an exact transformation. In the figure, the transitions a and b can be replaced by a set of transitions feeding a place. Similarly, the transition c can be replaced by a set of transitions coming out of the single place.

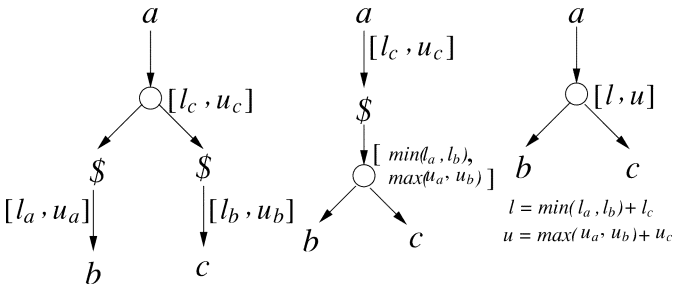


Fig. 12. Safe Transformation 4.

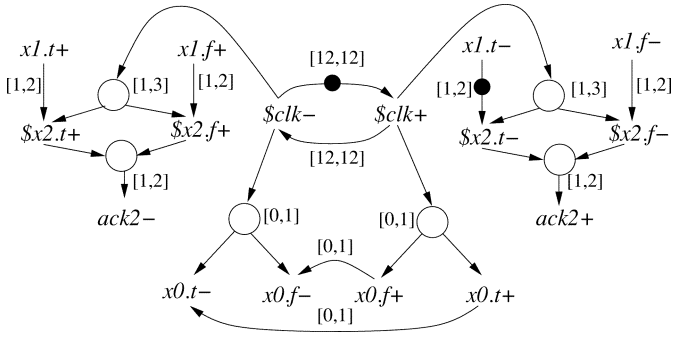


Fig. 13. Environment for Stage1 after removing *$ack3+* and *$ack3–* using Transformation 1 .

The fourth transformation involves a choice place and a pair of abstracted transitions, and it is depicted in Fig. 12. Similar to Transformation 3, if $l_a = l_b$ and $u_a = u_b$ then it is an exact transformation. These two transformations can be proved in a way similar to that used for Transformations 1 and 2 . Numerous other safe transformation have been developed and proved to be correct. More details can be found in [43].

Consider the TPN shown in Fig. 6. The abstracted transitions *$ack3+* and *$ack3–* can be removed using Transformation 1 resulting in the TPN shown in Fig. 13. Since *$x2.f+* and *$x2.t+* have the same place in their postsets, they can be merged together. Similarly, *$x2.f–* and *$x2.t–* can be merged, too. The TPN after applying Transformation 3 is shown in Fig. 14. Finally, *$x2.t+* and *$x2.t–* can be removed using Transformation 1 resulting in the final TPN for the environment to Stage1 shown in Fig. 15. This TPN can now be composed with the TPN for Stage1 shown in Fig. 3(a), and a timed state space exploration algorithm can be utilized to determine if Stage1 has any failures.

VII. Modular Verification

This section develops the theory necessary to justify an approach to modular verification that uses safe transformations. The theory is presented in terms of a circuit composed of two modules, but it can be readily extended to an arbitrary number of modules. The two modules are described by two TPNs, N_1 and N_2, and their corresponding trace structures, $\mathcal{T}(N_1) = \mathcal{T}_1 = \langle \mathcal{I}_1, \mathcal{O}_1, \mathcal{S}_1, \mathcal{F}_1 \rangle$ and $\mathcal{T}(N_2) = \mathcal{T}_2 = \langle \mathcal{I}_2, \mathcal{O}_2, \mathcal{S}_2, \mathcal{F}_2 \rangle$. The composition of $\mathcal{T}_1$ and $\mathcal{T}_2$ defines the behavior of the circuit which is represented by the trace structure $\mathcal{T}$ (i.e., $\mathcal{T} = \mathcal{T}_1 \parallel$

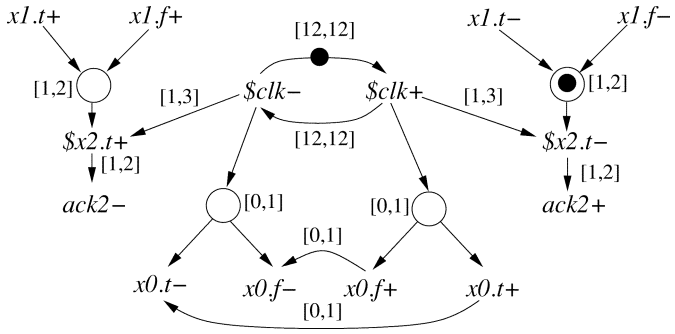


Fig. 14. Environment for Stage1 after removing *$x2.f+* and *$x2.f–* using Transformation 3.

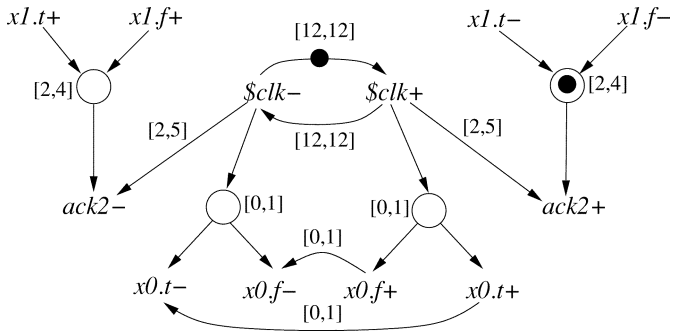


Fig. 15. Environment for Stage1 after removing *$x2.t+* and *$x2.t–* using Transformation 1 .

$\mathcal{T}_2$). The possible trace sets of $\mathcal{T}_1$ and $\mathcal{T}_2$ are denoted by $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Suppose $\mathcal{D}_1$ and $\mathcal{D}_2$ are the sets of internal signal transitions of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively (i.e., $\mathcal{D}_1 = \mathcal{O}_1 - \mathcal{I}_2$ and $\mathcal{D}_2 = \mathcal{O}_2 - \mathcal{I}_1$). The composition of $\mathcal{T}_1$ and $\mathcal{T}_2$ is only defined when the outputs of the two modules are disjoint (i.e., $\mathcal{O}_1 \cap \mathcal{O}_2 = \emptyset$). This implies that the internal signal transition sets are also disjoint (i.e., $\mathcal{D}_1 \cap \mathcal{D}_2 = \emptyset$).

The theory in this section allows us to verify the complete circuit by verifying each module individually. If both modules are correct, the complete circuit is also correct. When verifying the module $\mathcal{T}_1$, the module $\mathcal{T}_2$ is regarded as the environment for $\mathcal{T}_1$. Therefore, the internal signal transitions of $\mathcal{T}_2$ can be removed and the result of the verification is not affected. When $\mathcal{T}_2$ is chosen, a similar process is applied to $\mathcal{T}_1$. The idea is formulated in the following theorem.

Theorem 7.1: Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be internal signal transition sets of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. If $\mathcal{T}_1 \parallel \mathbf{hide}(\mathcal{D}_2)(\mathcal{T}_2)$ is failure-free and $\mathbf{hide}(\mathcal{D}_1)(\mathcal{T}_1) \parallel \mathcal{T}_2$ is failure-free, then $\mathcal{T} = \mathcal{T}_1 \parallel \mathcal{T}_2$ is also failure-free.

Proof: First, the failure set of $\mathcal{T}_1 \parallel \mathcal{T}_2$ can be found from the definition of composition

$$\mathcal{F} = ((\mathbf{del}^{-1}(\mathcal{D}_2)(\mathcal{F}_1)) \cap (\mathbf{del}^{-1}(\mathcal{D}_1)(\mathcal{P}_2))) \cup ((\mathbf{del}^{-1}(\mathcal{D}_1)(\mathcal{F}_2)) \cap (\mathbf{del}^{-1}(\mathcal{D}_2)(\mathcal{P}_1)))$$

where $\mathcal{F}_1 = \mathbf{fail}(\mathcal{P}_1)$ and $\mathcal{F}_2 = \mathbf{fail}(\mathcal{P}_2)$.

To prove this theorem, we must show that $\mathcal{F}$ is empty. This is achieved by using the fact that $\mathcal{T}_1 \parallel \mathbf{hide}(\mathcal{D}_2)(\mathcal{T}_2)$ is failure-free to show that the left-hand side of the union operator is empty. Similarly, the fact that $\mathbf{hide}(\mathcal{D}_1)(\mathcal{T}_1) \parallel \mathcal{T}_2$ is failure-free is used to show that the right-hand side is also empty.

First, suppose $\mathcal{T}_1 \parallel \mathbf{hide}(\mathcal{D}_2)(\mathcal{T}_2)$ is failure-free. This means that its failure set given below is empty

$$(\mathcal{F}_1 \cap \mathbf{del}^{-1}(\mathcal{D}_1)(\mathbf{del}(\mathcal{D}_2)(\mathcal{P}_2))) \cup (\mathcal{P}_1 \cap \mathbf{del}^{-1}(\mathcal{D}_1)(\mathbf{fail}(\mathbf{del}(\mathcal{D}_2)(\mathcal{P}_2)))) = \emptyset.$$

This comes from the definitions of **hide** and composition. Since $A \cup B = \emptyset \Rightarrow A = \emptyset$, the following is true

$$\mathcal{F}_1 \cap \mathbf{del}^{-1}(\mathcal{D}_1)(\mathbf{del}(\mathcal{D}_2)(\mathcal{P}_2)) = \emptyset. \tag{12}$$

From (4), a set of traces remains the same if it is applied to by the inverse delete and then the delete function on the same set of transitions. Applying this property with the transition set $\mathcal{D}_2$ to $\mathcal{F}_1$, (12) is transformed to

$$\mathbf{del}(\mathcal{D}_2)(\mathbf{del}^{-1}(\mathcal{D}_2)(\mathcal{F}_1)) \cap \mathbf{del}^{-1}(\mathcal{D}_1)(\mathbf{del}(\mathcal{D}_2)(\mathcal{P}_2)) = \emptyset. \tag{13}$$

From (3), a set of traces remains the same after exchanging the order of the inverse delete and the delete function if they use disjoint sets of transitions. After applying (3) to $\mathbf{del}^{-1}(\mathcal{D}_1)(\mathbf{del}(\mathcal{D}_2)(\mathcal{P}_2))$, (13) is transformed to

$$\mathbf{del}(\mathcal{D}_2)(\mathbf{del}^{-1}(\mathcal{D}_2)(\mathcal{F}_1)) \cap \mathbf{del}(\mathcal{D}_2)(\mathbf{del}^{-1}(\mathcal{D}_1)(\mathcal{P}_2)) = \emptyset.$$

Now both sides of the intersection operator have the delete function in the front. If $\mathbf{del}(\mathcal{D}_2)$ is extracted using (5), then we can determine the following:

$$\mathbf{del}(\mathcal{D}_2)((\mathbf{del}^{-1}(\mathcal{D}_2)(\mathcal{F}_1)) \cap (\mathbf{del}^{-1}(\mathcal{D}_1)(\mathcal{P}_2))) \subseteq \mathbf{del}(\mathcal{D}_2)(\mathbf{del}^{-1}(\mathcal{D}_2)(\mathcal{F}_1)) \cap \mathbf{del}(\mathcal{D}_2)(\mathbf{del}^{-1}(\mathcal{D}_1)(\mathcal{P}_2)) = \emptyset.$$

Since a subset of the empty set is also an empty set, the following holds:

$$\mathbf{del}(\mathcal{D}_2)((\mathbf{del}^{-1}(\mathcal{D}_2)(\mathcal{F}_1)) \cap (\mathbf{del}^{-1}(\mathcal{D}_1)(\mathcal{P}_2))) = \emptyset. \tag{14}$$

Finally, from (2), if the delete function is applied to an empty set, the resulting set is still empty, and vice versa. Therefore, after removing $\mathbf{del}(\mathcal{D}_2)$ from (14), we have the following:

$$(\mathbf{del}^{-1}(\mathcal{D}_2)(\mathcal{F}_1)) \cap (\mathbf{del}^{-1}(\mathcal{D}_1)(\mathcal{P}_2)) = \emptyset. \tag{15}$$

This proves the first half of the theorem. Now, we prove the second half. Suppose $\mathbf{hide}(\mathcal{D}_1)(\mathcal{T}_1) \parallel \mathcal{T}_2$ is failure-free. Thus, its failure set is

$$(\mathcal{P}_2 \cap \mathbf{del}^{-1}(\mathcal{D}_2)(\mathbf{fail}(\mathbf{del}(\mathcal{D}_1)(\mathcal{P}_1)))) \cup (\mathcal{F}_2 \cap \mathbf{del}^{-1}(\mathcal{D}_2)(\mathbf{del}(\mathcal{D}_1)(\mathcal{P}_1))) = \emptyset.$$

By applying the same steps above to the following:

$$(\mathcal{F}_2 \cap \mathbf{del}^{-1}(\mathcal{D}_2)(\mathbf{del}(\mathcal{D}_1)(\mathcal{P}_1))) = \emptyset$$

we can derive the following:

$$(\mathbf{del}^{-1}(\mathcal{D}_1)(\mathcal{F}_2)) \cap (\mathbf{del}^{-1}(\mathcal{D}_2)(\mathcal{P}_1)) = \emptyset. \tag{16}$$

The union of (15) and (16) is the failure set of $\mathcal{T}$. Since both (15) and (16) are empty, the failure set of $\mathcal{T} = \mathcal{T}_1 \parallel \mathcal{T}_2$ is also

empty. This theorem is naturally extended to a circuit consisting of more than two modules. ■

Determining if $\mathcal{T}_1 \parallel \mathbf{hide}(\mathcal{D}_2)(\mathcal{T}_2)$ and $\mathbf{hide}(\mathcal{D}_1)(\mathcal{T}_1) \parallel \mathcal{T}_2$ are failure-free is still an exponential problem, since the **hide** operator requires all the traces to be first derived then transitions are deleted from them. To address this problem, our method instead applies abstraction and safe transformations to the corresponding TPN to remove internal signal transitions, then the state space is explored to generate the new trace structure. Given a TPN N and its corresponding trace structure $\mathcal{T}$, from Lemma 5.1 we know that $\mathbf{hide}(\mathcal{D})(\mathcal{T})$ conforms to $\mathcal{T}(\mathbf{abs}(\mathcal{D})(N))$ (see Definition 4.1). Suppose N_1 and N_2 are the TPNs for $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. Therefore, combined with Lemma 4.2, we know that $\mathcal{T}_1 \parallel \mathbf{hide}(\mathcal{D}_2)(\mathcal{T}_2)$ conforms to $\mathcal{T}_1 \parallel \mathcal{T}(\mathbf{abs}(\mathcal{D}_2)(N_2))$ and $\mathbf{hide}(\mathcal{D}_1)(\mathcal{T}_1) \parallel \mathcal{T}_2$ conforms $\mathcal{T}(\mathbf{abs}(\mathcal{D}_1)(N_1)) \parallel \mathcal{T}_2$. Using these observations, we can prove the key result of this paper.

Theorem 7.2: Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be internal signal transition sets of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. If $\mathcal{T}_1 \parallel \mathcal{T}(\mathbf{abs}(\mathcal{D}_2)(N_2))$ is failure-free, and $\mathcal{T}(\mathbf{abs}(\mathcal{D}_1)(N_1)) \parallel \mathcal{T}_2$ is failure-free, then $\mathcal{T} = \mathcal{T}_1 \parallel \mathcal{T}_2$ is also failure-free.

Proof: From Lemma 5.1, we have

$$\mathbf{hide}(\mathcal{D}_1)(\mathcal{T}_1) \preceq \mathcal{T}(\mathbf{abs}(\mathcal{D}_1)(N_1))$$
$$\mathbf{hide}(\mathcal{D}_2)(\mathcal{T}_2) \preceq \mathcal{T}(\mathbf{abs}(\mathcal{D}_2)(N_2)).$$

From Lemma 4.2, we have

$$\mathbf{hide}(\mathcal{D}_1)(\mathcal{T}_1) \parallel \mathcal{T}_2 \preceq \mathcal{T}(\mathbf{abs}(\mathcal{D}_1)(N_1)) \parallel \mathcal{T}_2$$
$$\mathcal{T}_1 \parallel \mathbf{hide}(\mathcal{D}_2)(\mathcal{T}_2) \preceq \mathcal{T}_1 \parallel \mathcal{T}(\mathbf{abs}(\mathcal{D}_2)(N_2)).$$

Since $\mathcal{T}(\mathbf{abs}(\mathcal{D}_1)(N_1)) \parallel \mathcal{T}_2$ and $\mathcal{T}_1 \parallel \mathcal{T}(\mathbf{abs}(\mathcal{D}_2)(N_2))$ are failure-free, then $\mathbf{hide}(\mathcal{D}_1)(\mathcal{T}_1) \parallel \mathcal{T}_2$ and $\mathcal{T}_1 \parallel \mathbf{hide}(\mathcal{D}_2)(\mathcal{T}_2)$ are failure-free. From Theorem 7.1, $\mathcal{T}_1 \parallel \mathcal{T}_2$ is failure-free. ■

VIII. Experimental Results

We have incorporated our abstraction technique into our specification compiler [44] front-end to the ATACS tool. When verifying a system, it is supplied to ATACS with its environment, and the user indicates a component in the system to be verified. Before state space exploration, ATACS finds the environment for the component and the internal signals in the environment based on the interface of the component and removes as many transitions on those signals as possible using the abstraction technique described above. This process is repeated for all components in the system which have not been verified. The default is that all safe transformations introduced in Section VI are applied. However, the user can restrict the transformations to avoid a false negative. This section describes the application of our method to three timed circuit benchmarks. BAP, an enhanced version of the POSET timing analysis algorithm [45] is used as the underlying timing analysis engine for all examples in this section.

The first example is a multiple stage controller for a self-timed FIFO from Sun Microsystems. In [46], a highly optimized hand designed timed circuit implementation is presented. The

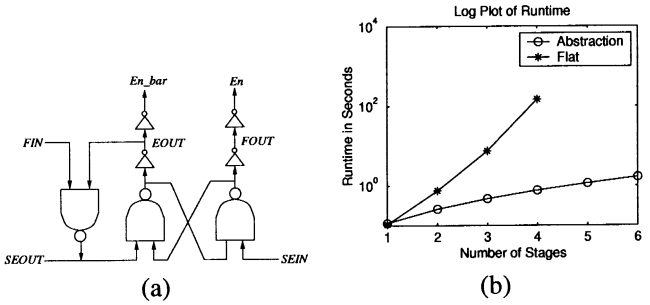


Fig. 16. (a) Control circuit for a single stage FIFO and (b) runtime result for a FIFO up to six stages.

circuit is shown in Fig. 16(a). The operation of the FIFO is as follows: whenever a stage that is *Full* is followed by a stage that is *Empty*, the data in the full stage is moved to the empty stage and the states of both stages are changed correspondingly. When a request comes in (*FIN*+) and the FIFO is empty (*EOUT* is high), the data is latched (*En_bar*+ and *En*–). In parallel, the insertion is acknowledged (*SEOUT*–) and the next stage is requested to accept the data (*FOUT*+). When the next stage accepts the data (*SEIN*–), the FIFO is set to be empty (*EOUT*+) and the latch is opened (*En_bar*– and *En*+).

The correctness of this circuit is highly dependent on timing parameters. We assume a delay of 90 to 110 time units for each gate. We assume that *FIN*+ transitions on the far left side of the FIFO can arrive anytime after 180 time units from the *SEOUT*+ transition, which is initially enabled. The transition *FIN*– occurs between 180 and 260 time units after *SEOUT*–. The transition *SEIN*– on the far right side goes low anytime after 90 time units from the *FOUT*+ transition, while *SEIN*+ occurs 90 to 110 time units after *FOUT*–. Without using abstraction, ATACS can only analyze a FIFO with up to four stages. For the FIFO with five stages, we had to kill the process after it ran for over a day. With abstraction on, however, ATACS easily proceeds to 100 stages. The verification at 100 stages takes approximately 31 min and 13 MB of memory. Fig. 16(b) shows the comparative runtime results up to four stages. The runtimes are taken from a 650-MHz Pentium III with 576 MB of memory. This same machine is used to generate all results in the remainder of this presentation.

The second example is the STARI communication circuit presented earlier and described in detail in [31]. In [27], the authors state that COSPAN, which uses a region technique for timing verification [47], ran out of memory attempting to verify a three-stage gate-level version of STARI on a machine with 1 GB of memory. This paper goes on to describe an abstract model developed by hand for STARI for which they could verify eight stages in 92.4 MB of memory and 1.67 h. Our automated abstraction method verifies a 15-stage STARI with a maximum memory usage of 32 MB of memory for a single stage in just over 27 min. Fig. 17 shows the comparative runtimes for verification with and without abstraction on STARI. As shown in the chart, BAP can verify STARI with up to 12 stages using 197 MB of memory in about 15.5 min. In the first few stages, the runtime for verification with abstraction is worse because abstraction itself takes time. The figure shows the runtime for flat analysis in

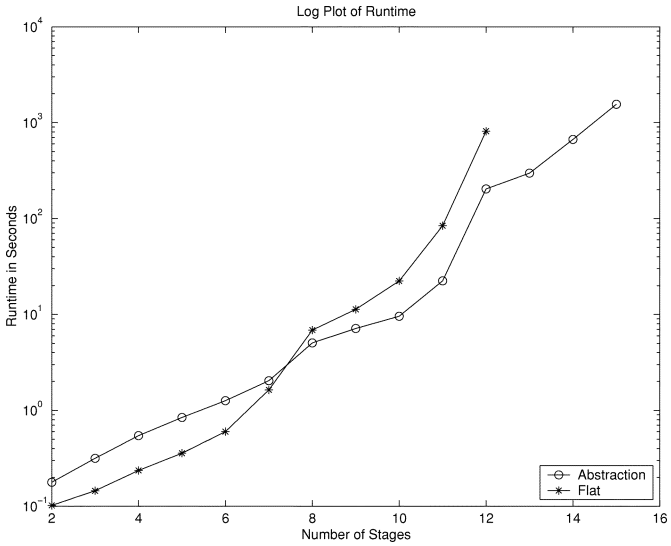


Fig. 17. Runtimes for STARI.

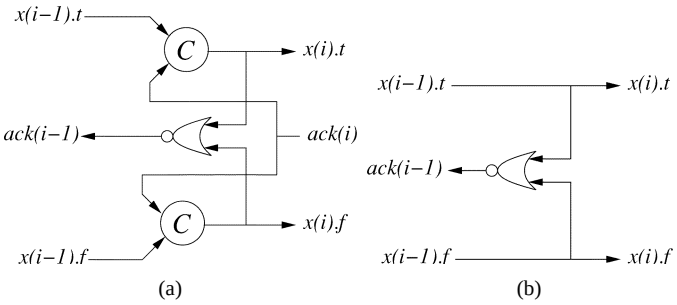


Fig. 18. Different implementations of STARI.

BAP to increase at a higher rate than the comparative runtime of BAP with abstraction in this example.

It is important to note that our approach is not constrained to a particular state space search algorithm. This is key because the abstraction method can be used in conjunction with any analysis algorithm. In [48], for example, an analysis algorithm is presented that can verify STARI up to 18 stages. If we used this analysis algorithm, then we would likely see improvement over our current results.

An interesting by-product of our approach on STARI is that the modular verification leads to circuit optimization. A traditional approach designs and verifies a single STARI stage and replicates it up to the desired size of the FIFO. An implementation of a STARI stage using the timing numbers in Fig. 4 is shown in Fig. 18(a). A STARI consisting of this stage implementation is verified correct. However, different implementations of STARI satisfying the same timing properties are available. These alternate implementations are discovered by the modular analysis. In an eight-stage STARI, for example, the C-elements in the first three stages used to store the data can be reduced to wires as shown in Fig. 18(b). In the last three stages, a generalized C-element using one less transistor can be used. Only the middle two stages require full C-elements. 80 literals and 160 transistors are required to implement an eight-stage STARI consisting of the same stages, while the second implementation requires 56 literals and 136 transistors. This example shows that verification using abstraction can help identify the redundant components in a circuit and simplify the design.

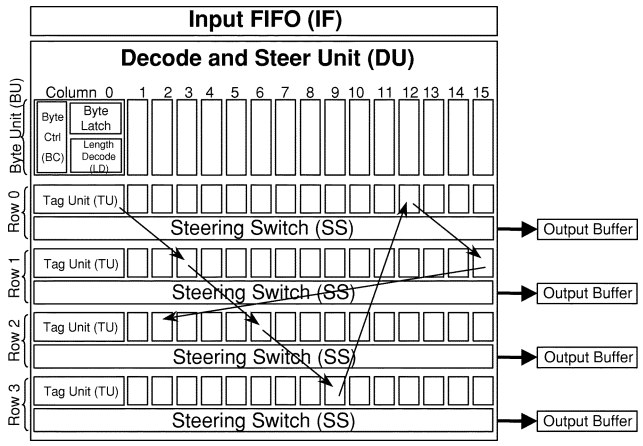


Fig. 19. RAPPID microarchitecture.

The last example comes from Intel's RAPPID circuit. The goal of this example is to demonstrate the application of our method on a timed circuit with irregular structures. The RAPPID circuit is a fully asynchronous instruction-length decoder for the Pentium II 32-bit MMX instruction set. In this instruction set, each instruction can be from 1 to 15 bytes 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 microarchitecture is shown in Fig. 19. The RAPPID decoder reads in a 16-byte cache line, and it decodes each byte as if it is the first byte of a new instruction. Each byte speculatively determines the length of an instruction beginning with this byte. The actual first byte of the current instruction is marked with a tag. This byte uses the length that it determined to decide which byte is the first byte of the next instruction. It then signals that byte while notifying all bytes in between to cancel their length calculations and forward the bytes of the current instruction to an output buffer. To improve performance, four rows of tag units and output buffers are used in a round-robin four-issue fashion.

The RAPPID design achieved a significant performance improvement over its synchronous counterpart using circuits that are aggressively optimized through timing assumptions. For example, the tag unit circuit shown in Fig. 20 requires timing assumptions for correct operation. In typical asynchronous communication, a request is transmitted followed by an acknowledge being received to indicate that the circuit can reset. In this case, there is no explicit acknowledgment, but rather, acknowledgment comes by way of a timing assumption. Once a tag arrives (i.e., *TagArrived* is high), if the instruction and steering switch are ready (i.e., *InstRdy* and *SSRdy* are high), then the course is set to begin to reset *TagArrived*. The result is that the signal produced on $TagOut_i$ is a pulse. The shape of the $TagOut_i$ pulse must be carefully controlled for the circuit to operate correctly.

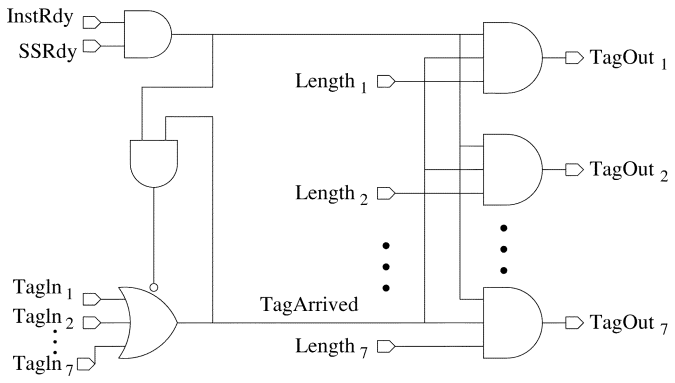


Fig. 20. Tag unit circuit.

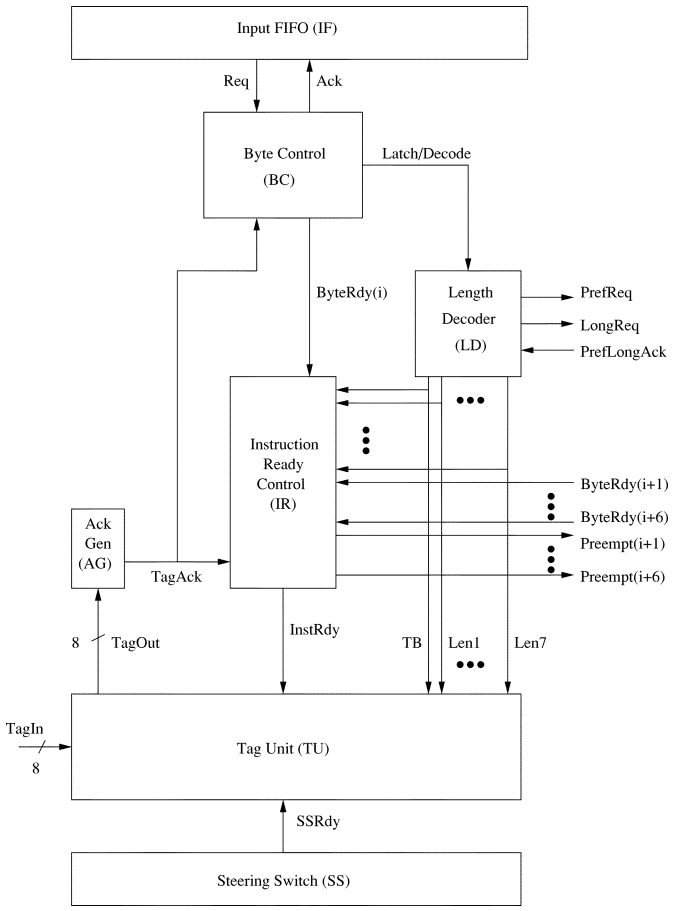


Fig. 21. Block diagram for the RAPPID control circuit.

In this example, we analyze a somewhat simplified, although still nonregular, version of the control portion for a column of the RAPPID circuit depicted in Fig. 21. It is simplified in that signals which would preempt this column have not been incorporated. Data has also been abstracted away as modeling the length calculation using a Petri net would certainly be quite complex, and it is sufficient to represent this using nondeterminism.

The complexity of this simplified model of a column from RAPPID is significant. The TPN description of the column has 115 transitions on 49 signal wires. Flat analysis runs out of memory in about an hour on our test machine (the stack depth was in excess of 27 000 entries and climbing indicating that it had a long way to go). Our approach considers each of the major control modules individually: TU, AG, BC, and IR. The results of the analysis are shown in Table I. The TU, AG, and BC units are rapidly verified within a minute of total runtime for all three modules. The IR module is further decomposed into seven modules due to its size. Modules $(\mathrm{PE1}, \ldots, \mathrm{PE6})$ generate the *Preempt* signals, and module IR1 generates the *InstRdy* signal. The analysis of all 10 modules requires 16 runs, which on average, reduce the TPN by over half. The entire analysis also explores on average less than 1600 timed states and 1000 untimed states, and uses under 20 megabytes of memory per verification run. The total analysis time is less than 11 minutes for this example.

TABLE I
RESULTS FOR RAPPID EXAMPLE

Unit	Trans.	Zones	States	Mem.(MB)	Time(s)
RAPPID	115	>142,000	>141,000	>576	>1 hour
TU	80	2689	1923	17.5	27.1
AG	64	154	154	14.4	7.9
BC	54	494	342	16.6	11.7
PE1 (run 1)	43	546	378	15.2	13.9
PE1 (run 2)	54	434	322	14.9	10.3
PE2 (run 1)	43	546	378	15.2	13.9
PE2 (run 2)	54	434	322	14.9	10.3
PE3 (run 1)	43	546	378	15.2	13.9
PE3 (run 2)	54	434	322	14.9	10.3
PE4 (run 1)	43	546	378	15.2	13.9
PE4 (run 2)	54	434	322	14.9	10.3
PE5 (run 1)	43	546	378	15.2	13.9
PE5 (run 2)	54	434	322	14.9	10.3
PE6 (run 1)	43	546	378	15.2	13.9
PE6 (run 2)	54	434	322	14.9	10.3
IR1	66	15364	7888	61.3	426.4
Avg/Total	53	1536	907	18.2	618.3

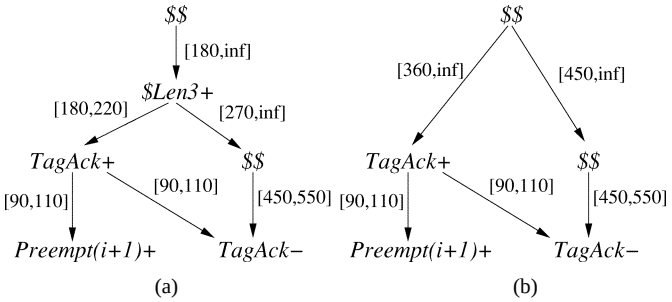


Fig. 22. Reason for false negatives in PEi units.

The reason that each of the PEi units are run twice is because the first analysis run finds a false negative. The introduction of the false negative can be illustrated using a snippet of the TPN model before and after applying a transformation as shown in Fig. 22(a) and (b). Before the transformation [see Fig. 22(a)] it is clear that the only possible trace of visible signal transitions is: *TagAck+*, $Preempt(i+1)+$, *TagAck–*. $Preempt(i+1)+$ must occur within 330 time units after *$Len3+*, while *TagAck–* cannot occur until at least 720 time units after *$Len3+*. After Transformation 2 removes *$Len3+* [see Fig. 22(b)], however, the trace *TagAck+*, *TagAck–*, $Preempt(i+1)+$ is introduced. This additional trace eventually results in a failure in our model. After obtaining a false negative, we rerun the abstraction and analysis without using Transformation 2. Although more transitions remain in the net to be analyzed (54 as opposed to 43 in this case), the analysis time for both runs is still less than 25 s.

A final point to mention regarding the results for the analysis of RAPPID is that the runtimes include more than the cost of state space exploration. The reported runtimes are a composite of state space exploration, correctness checking, and synthesis to generate a circuit implementation for the module under analysis. The modular approach described in this paper is not limited to verification, but can be also applied to improving synthesis time for timed circuits. State based synthesis relies critically on state space exploration. In all cases, the synthesized circuit is either the same as that reported for RAPPID (such as the one shown in Fig. 20) or a version appropriate to the simplifying assumptions described earlier.

IX. Conclusion

Since the size of state space grows exponentially in the size of a design, state explosion is the most serious challenge for any verification method based on state space exploration. This paper describes a theoretical framework and techniques to avoid state space explosion encountered in large designs by partitioning a design into modules with constrained complexity and verifying each module separately. This paper proves that the entire design can be shown to be correct if each of its constituent modules is correct. An abstraction technique is described to construct the environment for each module, and the concept of safe transformations is used to derive an abstraction for that environment. This paper also presents several transformations that are proven to satisfy the requirement of safe transformations.

We have performed case studies on three examples including an industrial scale design. Our results are promising and show that our abstraction method is faster and uses less memory than flat analysis for these designs. This enables the verification of systems that are beyond the reach of flat analysis. We have also shown that although false negative results are possible, they are rare in our case studies. Moreover, an incremental back off from the applied transformations can be efficiently implemented to remove false negatives. Another important point is that our method can be built on top of any reachability analysis algorithm. If some better reachability analysis algorithm is available, then it can be combined with our abstraction method to extend capacity to even larger designs. In particular, our preliminary analysis has shown that combining abstraction with a partial order based analysis technique can bring even further improvements.

Currently, our method requires indication from users to select modules to verify. In the future, we plan to develop a partition approach to automatically break a design into a set of modules each with optimal complexity. We are also looking at methods to better treat false negatives. In the current approach, when verification fails, we rerun the analysis by incrementally undoing the transformations until we establish the truthfulness of the violation. The current implementation is brute force. In the future, we would like to develop techniques to automatically analyze the result and determine which transformation causes the failure.

Acknowledgment

The authors would like to thank W. Belluomini of IBM and T. Yoneda of the Tokyo Institute of Technology for their helpful comments. They would also like to thank the anonymous reviewers that provided numerous helpful comments and suggestions which greatly improved the presentation of this paper.

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*, pp. 66–74, May-June 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, pp. 217–228, Feb. 2001.

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

[4] J. R. Burch, "Modeling timing assumptions with trace theory," in *Proc. Int. Conf. Comput. Design*, 1989, pp. 208–211.

[5] J. Burch, "Delay models for verifying speed-dependent asynchronous circuits," in *ACM Int. Workshop Timing Issues Specification Synthesis Digital Syst.*, Mar. 1992.

[6] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[7] D. Dill, S. Nowick, and R. Sproull, "Specification and Automatic Verification of Self-Timed Queues," Stanford Univ., Stanford, CA, Tech. Rep. CSL-TR-89-387, 1989.

[8] W. Belluomini and C. J. Myers, "Timed state space exploration using posets," *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 501–520, May 2000.

[9] ——, "Timed circuit verification using tel structures," *IEEE Trans. Computer-Aided Design*, vol. 20, pp. 129–146, Jan. 2001.

[10] 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 South Pacific Design Automation Conf.*, Feb. 2001, pp. 335–340.

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

[12] T. G. Rokicki, "Representing and Modeling Circuits," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1993.

[13] T. G. Rokicki and C. J. Myers, "Automatic verificaton of timed circuits," in *Proc. Int. Conf. Computer-Aided Verification*, 1994, pp. 468–480.

[14] S. Yovine, "KRONOS: A verification tool for real-time systems," *Springer Int. J. Software Tools Technol. Transfer*, vol. 1, no. 1/s2, Oct. 1997.

[15] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL – A tool suite for automatic verification of real-time systems," in *Hybrid Syst.*, 1995, vol. 1066, LNCS, pp. 232–243.

[16] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Inform. Computat.*, vol. 98, no. 2, pp. 142–170, 1998.

[17] K. McMillan, "Symbolic Model Checking: An Approach to the State Explosion Problem," Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, 1993.

[18] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677–691, Aug. 1986.

[19] M. Bozga, O. Maler, A. Pnueli, and S. Yovine, "Some progress in the symbolic verification of timed automata," in *Int. Conf. Computer-Aided Verification*, vol. 1254, LNCS, 1997.

[20] R. A. Thacker, W. Belluomini, and C. J. Myers, "Timed circuit synthesis using implicit methods," in *Proc. Int. Conf. VLSI Design*, 1999, pp. 181–188.

[21] A. Valmari, "A stubborn attack on state explosion," in *Proc. Int. Conf. Computer-Aided Verification*, vol. 531, LNCS, June 1990, pp. 176–185.

[22] P. Godefroid, "Using partial orders to improve automatic verification methods," in *Proc. Int. Conf. Computer-Aided Verification*, vol. 531, LNCS, 1990, pp. 176–185.

[23] K. McMillan, "Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits," in *Proc. Int. Workshop Computer-Aided Verification*, vol. 663, LNCS, G. v. Bochman and D. K. Probst, Eds., 1992, pp. 164–177.

[24] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi, "Partial order reductions for timed systems," in *Proc. Int. Conf. Concurrency Theory*, 1998, pp. 485–500.

[25] O. Grumberg and D. Long, "Model checking and modular verification," *ACM Trans. Programm. Lang. Syst.*, vol. 16, pp. 843–872, 1994.

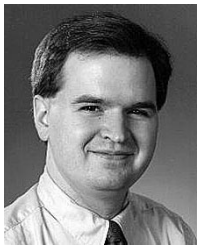
[26] S. Tasiran, R. Alur, R. Kurshan, and R. Brayton, "Verifying abstractions of timed systems," in *Proc. 7th Int. Conf. Concurrency Theory*, vol. 1119, LNCS, 1996, pp. 546–562.
[27] S. Tasiran and R. K. Brayton, "Stari: A case study in compositional and heirarchical timing verification," in *Proc. Int. Conf. Computer-Aided Verification*, vol. 1254, LNCS, 1997.
[28] E. Clarke, O. Grumberg, and D. Long, "Model checking and abstraction," *ACM Trans. Programm. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1994.
[29] D. Dams, R. Gerth, and O. Grumberg, "Abstract interpretation of reactive systems," *ACM Trans. Programm. Lang. Syst.*, vol. 19, no. 2, pp. 253–291, 1997.
[30] C. Ramchandani, "Analysis of Asynchronous Concurrent Systems by Timed Petri Nets," Mass. Inst. Technol., Cambridge, MA, Tech. Rep. Project MAC Tech. Rep. 120, 1974.
[31] M. R. Greenstreet, Stari: Skew Tolerant Communication, 1997.
[32] C. A. R. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice-Hall, 1985.
[33] R. Milner, *Communication and Concurrency*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
[34] J. Snepscheut, *Trace Theory and VLSI Design*. New York: Springer-Verlag, 1985, vol. 200, LNCS.
[35] J. R. Burch, "Trace Algebra for Automatic Verification of Real-Time Concurrent Systems," Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, 1992.
[36] D. Dill, *Private Communications*. Stanford, CA: Stanford Univ., 2000.
[37] I. Suzuki and T. Murata, *Stepwise Refinements for Transitions and Places*. New York: Springer-Verlag, 1982.
[38] ——, "A method for stepwise refinements and abstractions of petri nets," *J. Comput. Syst. Sci.*, vol. 27, no. 1, pp. 51–76, 1983.
[39] G. Berthelot, *Checking Properties of Nets Using Transformations*. New York: Springer-Verlag, 1986, vol. 222, LNCS, pp. 19–40.
[40] R. Johnsonbaugh and T. Murata, "Additional methods for reduction and expansion of marked graphs," *IEEE Trans. Circuits Syst.*, vol. CAS-28, pp. 1009–1014, Oct. 1981.
[41] T. Murata, "Petri nets: Properties, analysis, and applications," *Proc. IEEE*, vol. 77, pp. 541–580, Apr. 1989.
[42] T. Murata and J. Y. Koh, "Reduction and expansion of live and safe marked graphs," *IEEE Trans. Cicruits Syst.*, vol. CAS-27, pp. 68–70, Jan. 1980.
[43] H. Zheng, "Automatic Abstraction for Synthesis and Verification of Timed Systems," Ph.D. dissertation, Univ. Utah, Salt Lake City, 2001.
[44] ——, "Specification and Compilation of Timed Systems," master's thesis, Univ. Utah, Salt Lake City, 1998.
[45] E. Mercer, C. Myers, and T. Yoneda, "Improved poset timing analysis in timed petri nets," in *Proc. 10th Workshop Synthesis Syst. Integration Mixed Technol.*, Oct. 2001.
[46] C. Molnar, I. Jones, B. Coates, and J. Lexau, "A FIFO ring oscillator performance experiment," in *Proc. Int. Symp. Adv. Res. Asynchronous Circuits Syst.*, Apr. 1997, pp. 279–289.
[47] R. Alur and R. P. Kurshan, "Timing analysis in cospan," in *Hybrid Syst. III: Verification and Control*. New York: Springer-Verlag, 1996, vol. 1066, LNCS, pp. 220–231.
[48] M. Bozga, O. Maler, and S. Tripakis, "Modeling and verification of the stari chip using timed automata," in *Proc. 10th IFIP WG10.5 Adv. Res. Working Conf. Correct Hardware Design Verification Methods*, vol. 1703, 1999, pp. 125–141.

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

He is a Research Scientist with the Microelectronics Division, IBM, Essex Junction, VT. His current interests are in the application of formal methods in the computer industry, devising abstraction techniques to increase the capability of model checking, and advanced architectures for low power and high performance.

Eric Mercer (M'97) received the Ph.D. degree in electrical engineering from the University of Utah, Salt Lake City, in 2002.

He is an Assistant Professor in computer science at Brigham Young University, Provo, Utah. His interests include timing analysis, real-time semiformal verification, and parallel algorithms, as well as low-power architectures for high-performance applications.



Chris Myers (S'91–M'96) 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 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* (New York: Wiley). He is also a coinventor of four patents. His current research interests are in 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.