

Hazard Checking of Timed Asynchronous Circuits Revisited

Frédéric Béal
Tokyo Institute of Technology
Tokyo, Japan
frederic@yt.cs.titech.ac.jp

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

Chris J. Myers
University of Utah
Salt Lake City, USA
myers@ece.utah.edu

Abstract

This paper proposes a new approach for the hazard checking of timed asynchronous circuits. Previous papers proposed either exact algorithms, which suffer from state-space explosion, or efficient algorithms which use a (conservative) approximation to avoid state-space explosion but can result in the rejection of designs which are valid. In particular, [7] presents a timed extension of the work in [1] which is very efficient but is not able to handle circuits with internal loops, which prevents its use in some cases. We propose a new approach to the problem in order to overcome the mentioned limitations, without sacrificing efficiency. To do so, we first introduce a general framework targeted at the conservative checking of safety failures. This framework is not restricted to the checking of timed asynchronous circuits. Secondly, we propose a new (conservative) semantics for timed circuits, in order to use the proposed framework for hazard checking of such circuits. Using this framework with the proposed semantics yields an efficient algorithm that addresses the limitations of the previous approaches.

1 Introduction

The main particularity of *timed asynchronous circuits* [6] is the presence of timing information, unlike the *speed independent* approach [5]. While the additional information makes it possible to improve the quality of the generated designs [11, 12], by reducing the number of gates, and by making technology mapping easier, it can significantly increase the computational complexity of the synthesis and verification tools.

In the course of the synthesis of asynchronous circuits, extra care must be taken to avoid *hazards*. Hazards are conditions in which unexpected order of propagation of signals causes incorrect behaviour in the design. This can happen because of delays related to the operation of the gates. There are two approaches to the problem of hazard checking in timed asynchronous circuits. On the one hand, there

is the exact verification approach [10, 14, 8, 2, 15]. Though, this approach suffers from high cost, making it impractical for large designs. Conservative methods for speed independent circuits are proposed in [1] which reduce the state explosion problem by restricting exploration to the specification state space. In [7], these methods are extended to timed circuits. These approaches, though, are restricted to a subclass of circuits, as they cannot handle *internal loops* which appear frequently in the state variable insertion process during the automatic synthesis of circuits, and are somewhat overly-conservative, rejecting designs which respect the specification. Other approaches include [13], which applies a very general framework (abstract interpretation: [4]) to the problem of verification of asynchronous circuits, and [3], which uses a more symbolic approach, trading execution speed for expressive power.

Section 2 presents a general framework for conservative representation of designs. This framework is general and not limited to the problem at hand (from this point of view, it is related to abstract interpretation, though it is less general than abstract interpretation itself and uses a more practical approach). Then, general notions about timed asynchronous circuits are introduced, including a new representation of the behaviour of circuits which is *compact* and *precise*, though conservative, in order to apply the general framework. In Section 4, the derived algorithm is presented. Its complexity is similar to the one of [7], and it overcomes the limitations of the approach in [7]. Those limitations include the impossibility to check circuits that have internal loops and to detect that some hazards may not propagate to primary outputs. The proposed algorithm solves these limitations in a satisfactory manner. Finally, Section 5 shows some experimental results to demonstrate the proposed method.

2 A new framework

Before describing the framework itself, a few definitions are necessary.

2.1 Definitions

An *alphabet* is a finite set, the elements of which are called letters. A *word* is a finite sequence of letters. $w \cdot w'$ denotes the concatenation of the words w and w' ; ε is the empty word. If A and B are two set of words, then $A + B$ represents the set $A \cup B$, $A \cdot B$ represents the set of words $w \cdot w'$ where $w \in A$ and $w' \in B$, A^+ is the set of words $w_1 \cdot \dots \cdot w_n$ where $w_i \in A$, and $A^* = A^+ + \{\varepsilon\}$. The *projection* of a word w on a set A (denoted $\Pi_A(w)$) is the word obtained by removing from w the letters that do not appear in A .

A state machine \mathcal{M} is a 4-tuple $\mathcal{M} = (\Phi, \phi_0, X, T)$ where Φ is a set of states, $\phi_0 \in \Phi$ is the initial state, X is an alphabet and $T \subseteq \Phi \times X \times \Phi$ is a transition relation. Let $\mathbf{States}(\mathcal{M})$ and $\mathbf{Trans}(\mathcal{M})$ denote Φ and T , respectively. If $(\phi, x, \phi') \in T$, the notation $\mathcal{M} : \phi \rightarrow^x \phi'$ is used (or $\phi \rightarrow^x \phi'$ when there is no possible confusion). This notation is extended to $\phi \rightarrow^w \phi'$ for $w \in X^*$. A state machine \mathcal{M} is said to be *deterministic* if and only if for all $\phi, \phi', \phi'' \in \mathbf{States}(\mathcal{M})$ and $w \in X^*$, $\phi \rightarrow^w \phi'$ and $\phi \rightarrow^w \phi''$ imply $\phi' = \phi''$.

For a word w , $post^w(q)$ is the set $\{q' \mid q \rightarrow^w q'\}$. Similarly, for a set \mathcal{L} of words, $post^{\mathcal{L}}(q)$ is the set $\{q' \mid \exists w \in \mathcal{L}, q \rightarrow^w q'\}$.

Let \mathcal{M} and \mathcal{N} be two state machines on the same alphabet. A relation \mathcal{R} between states of \mathcal{M} and \mathcal{N} is a *simulation relation*, if and only if,

- \mathcal{R} is not empty,
- if $\mathcal{R}(\phi, q)$ and $x \in X$, with $\phi \rightarrow^x \phi'$, then there exists q' , such that $q \rightarrow^x q'$ and $\mathcal{R}(\phi', q')$.

where ϕ and ϕ' are states from \mathcal{M} and q and q' are states from \mathcal{N} .

Three distinct alphabets I, O and N are used (respectively *input*, *output*, and *internal* transitions¹). A *specification* state machine is a state machine on $I + O$. An *implementation* state machine is a state machine on $I + O + N$. The specification and implementation states machines are deterministic. In the rest of the section, \mathcal{S} is a specification state machine and \mathcal{I} is an implementation state machine.

The simulation relation is extended to the case where the left item is a specification state machine and the right item is an implementation state machine, by projecting out internal transitions on the implementation state machine, as follows: if q and q' in $\mathbf{States}(\mathcal{I})$, $x \in I + O$, then $q \rightsquigarrow^x q'$ denotes that there exists $u, v \in N^*$ with $\mathcal{I} : q \rightarrow^{u \cdot x \cdot v} q'$. Then, a simulation relation \mathcal{R} between the specification state machine \mathcal{S} and the implementation state machine \mathcal{I} must sat-

¹In general, a transition is a pair (*wire*, *direction*), often denoted by w^+ or w^- . However, in this section, transitions are represented by single letters.

isfy: if $\mathcal{R}(\phi, q)$, $\mathcal{S} : \phi \rightarrow^x \phi'$ with $x \in I + O$, then there exists q' , with $\mathcal{I} : q \rightsquigarrow^x q'$ and $\mathcal{R}(\phi', q')$.

It is supposed in the rest of this section that for the initial specification state ϕ_0 and the initial implementation state q_0 , there exists $\mathcal{R}(\phi_0, q_0)$. In other words, it is possible to emulate the execution of the specification in the implementation state space.

2.2 Proposed framework

$\mathcal{D}_{\mathcal{I}}$ denotes $2^{\mathbf{States}(\mathcal{I})}$. $\mathcal{D}_{\mathcal{S}} = \mathbf{States}(\mathcal{S})$ denotes the *specification domain*. This definition is based on the fact that a set of implementation states corresponds to one given specification state.

Definition 1 The *concretization* (Figure 1) $conc : \mathcal{D}_{\mathcal{S}} \rightarrow \mathcal{D}_{\mathcal{I}}$ is defined as follows:

$$conc(\phi) = \left\{ q \mid \exists w \in (I + O + N)^*, \begin{array}{l} \phi_0 \xrightarrow{\Pi_{I+O}(w)} \phi \\ q_0 \xrightarrow{w} q \end{array} \right\}$$

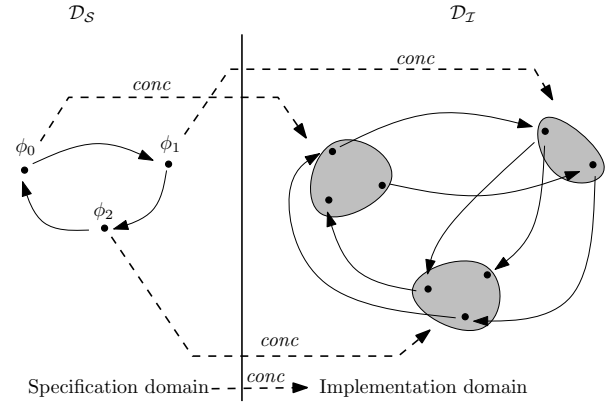


Figure 1. The concretization function.

The concretization of a specification state is the exact set of reachable implementation states corresponding to this particular specification state. An exact checking algorithm must compute this function, but the complexity of this is in general exponential or worse, so this should be avoided.

Using the specification in Figure 2 with the implementation given in Figure 3, the following holds :

$$\begin{aligned} conc(\phi_0) &= \{q_0\} \\ conc(\phi_1) &= \{q_1, q_2, q_3\} \\ conc(\phi_2) &= \{q_4\} \\ conc(\phi_3) &= \{q_5, q_6\} \end{aligned}$$

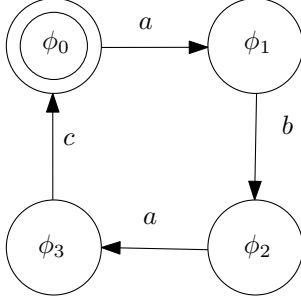


Figure 2. An example specification with $I = \{a\}$, $O = \{b, c\}$.

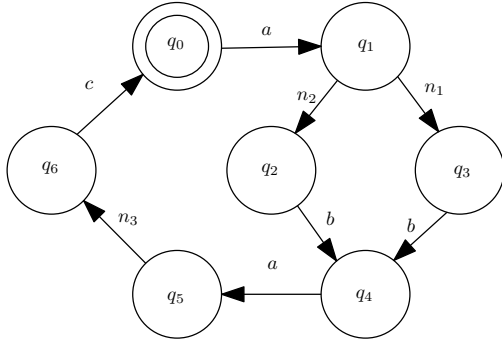


Figure 3. An example implementation with $I = \{a\}$, $O = \{b, c\}$, $N = \{n_1, n_2, n_3\}$.

Definition 2 The *closure under internal transitions* of the set $X \in \mathcal{D}_{\mathcal{I}}$ is the set

$$\alpha(X) = \bigcup_{q \in X} \text{post}^{N^*}(q)$$

As an example, considering Figure 3, where a is an input signal, b, c are output signals and the n_k are internal signals, $\alpha(\{q_1\}) = \{q_1, q_2, q_3\}$.

A set X is *closed under internal transitions* if $X = \alpha(X)$. The collection of closed sets is denoted by $\mathcal{D}_{\mathcal{I}}'$. Intuitively, $\mathcal{D}_{\mathcal{I}}'$ hides the internal transitions of the implementation, hence is expected to be easier to handle.

In our example,

$$\mathcal{D}_{\mathcal{I}}' = \left\{ \begin{array}{l} \{q_0\}, \{q_1, q_2, q_3\}, \\ \{q_2\}, \{q_3\}, \{q_4\}, \{q_5, q_6\}, \{q_6\} \end{array} \right\}.$$

Now, the definition of the transition relation on $\mathcal{D}_{\mathcal{I}}'$ is as follows: if $X, Y \in \mathcal{D}_{\mathcal{I}}'$ and $s \in I + O$, then $X \Rightarrow^s Y$ denotes that $Y = \alpha(\bigcup_{q \in X} \text{post}^s(q))$.

As for the example, the following transitions on $\mathcal{D}_{\mathcal{I}}'$: $\{q_0\} \Rightarrow^a \{q_1, q_2, q_3\} \Rightarrow^b \{q_4\} \Rightarrow^a \{q_5, q_6\} \Rightarrow^c \{q_0\}$

exist. The rest of the states in $\mathcal{D}_{\mathcal{I}}'$ are unreachable from $\{q_0\}$, so they are not interesting for our purpose. It can be remarked that the graph structure on the reachable states of $\mathcal{D}_{\mathcal{I}}'$ is isomorphic to the graph structure on $\mathcal{D}_{\mathcal{S}}$.

It can be proven that the alternative representation $\mathcal{D}_{\mathcal{I}}'$ is equivalent to $\mathcal{D}_{\mathcal{I}}$ in terms of computation of the reachable implementation states. This also proves that the new transition relation \Rightarrow is not easily computable. Hence, overapproximation is needed. The purpose of the proposed framework is to introduce a methodology to create such overapproximations, which is performed through the abstraction of the internal signals to have good performance (by limiting exploration to the specification state space).

For $X \in \mathcal{D}_{\mathcal{I}}'$, it is possible to define the *forward closure* of X , denoted by \overrightarrow{X} , as follows:

$$\overrightarrow{X} = \{w \in (O + N)^* \cdot (I + \varepsilon) \mid \exists q \in X, \text{post}^w(q) \neq \emptyset\}$$

This forward closure includes any sequence of internal and outputs transitions, followed eventually by an input transition, that can occur from X .

Using the example,

$$\begin{aligned} \overrightarrow{\{q_0\}} &= \{\varepsilon, a\} \\ \overrightarrow{\{q_1, q_2, q_3\}} &= \{\varepsilon, n_1, n_2, n_1b, n_2b, n_1ba, n_2ba, b, ba\} \\ \overrightarrow{\{q_4\}} &= \{\varepsilon, a\} \\ \overrightarrow{\{q_5, q_6\}} &= \{\varepsilon, n_3, n_3c, n_3ca, c, ca\} \end{aligned}$$

It is now possible to define the first object that is directly manipulated by the framework.

A *symbolic domain* is any set $\mathcal{D}_{\mathcal{I}}^{\sharp}$ such that any subset $A \subseteq \mathcal{D}_{\mathcal{I}}^{\sharp}$ has a least upper bound, denoted by $\sqcup A$, satisfying :

$$\forall a \in A, a \prec \sqcup A \quad (1)$$

$$(\forall a \in A, a \prec b) \Rightarrow \sqcup A \prec b, \quad (2)$$

where \prec represents an ordering relation, which is defined later. The alternative notations $\sqcup_{x \in X} x = \sqcup X$ and $a \sqcup b = \sqcup\{a, b\}$ are used in the rest of this paper. The elements of $\mathcal{D}_{\mathcal{I}}^{\sharp}$ are called *symbolic states*.

A *symbolic representation* is a pair $(\mathcal{D}_{\mathcal{I}}^{\sharp}, \text{base})$, where *base* is a function $\mathcal{D}_{\mathcal{I}}^{\sharp} \rightarrow \mathcal{D}_{\mathcal{I}}$. The symbolic representation should be chosen so as to be easy to manipulate. The function $\overrightarrow{\text{trans}} : \mathcal{D}_{\mathcal{I}}^{\sharp} \rightarrow 2^{(I+O+N)^*}$ is defined as $\overrightarrow{\text{trans}}(S) = \alpha(\text{base}(S))$. The set $\text{base}(S)$ is called the *base state* of S , and $\overrightarrow{\text{trans}}(S)$ represents the *set of enabled transitions* of S .

The ordering relation \prec between symbolic states is defined as :

$$S \prec S' \iff \text{base}(S) \subseteq \text{base}(S'). \quad (3)$$

The reason why the symbolic domain should have least upper bounds is as follows. During full state exploration, when visiting a specification state that has already been visited (for example, through a different path), the corresponding set of implementation states becomes the union of the previously computed set and the newly computed one, and exploration is continued. In order to obtain similar behaviour in the symbolic representation, the \sqcup operation is used. It follows from (3) that the least upper bound satisfies the intuitive requirement that, if $S'' = S \sqcup S'$, then any implementation state represented by S or S' is also represented by S'' .

Using the previous example, a particular symbolic representation can be defined, of which the $\mathcal{D}_{\mathcal{I}}^{\sharp}$ is $\{S_0, S_1, S_2, \Omega\}$ with $base(S_0) = \{q_0\}$, $base(S_1) = \{q_1, q_2, q_3, q_4\}$, $base(S_2) = \{q_5, q_6\}$ and $base(\Omega) = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$.

The symbolic representation is used to capture the behaviour of the implementation state machine at every specification state transition. Now, it is possible to define the operations that can be performed on the symbolic states, that is, the *forward implication* and *backward implication* functions, which are defined as follows:

Definition 3 A *forward implication* function is $forw : I \times \mathcal{D}_{\mathcal{I}}^{\sharp} \rightarrow \mathcal{D}_{\mathcal{I}}^{\sharp}$, satisfying the following property: for all $S \in \mathcal{D}_{\mathcal{I}}^{\sharp}$ and $x \in I$, if $S' = forw(x, S)$ then for all $q \in base(S)$ and $w = u \cdot x \in trans(S)$ with $u \in N^*$, $post^{u \cdot x}(q) \subseteq base(S')$.

Using the previous example : for $S = S_0$ and $x = a$, then $trans(S) = \{\epsilon, a\}$, and the condition is that q_1 must belong to $base(S')$. By definition, this gives $S' = S_1$ or $S' = \Omega$. Note that Ω is used to represent unknown behaviour. That is why $base(\Omega)$ is the set of all implementation states.

Definition 4 A *backward implication* function is $back : O \times \mathcal{D}_{\mathcal{I}}^{\sharp} \rightarrow \mathcal{D}_{\mathcal{I}}^{\sharp}$, satisfying the following property: for all $S \in \mathcal{D}_{\mathcal{I}}^{\sharp}$ and $x \in O$, if $S' = back(x, S)$, then if for some $q \in base(S)$, $u \cdot x \cdot v \in trans(S)$ with $u, v \in N^*$ and $q \xrightarrow{u \cdot x} q'$, then $q' \in base(S')$.

Using the same example, for $S = S_1$ and $x = b$, we obtain : q_4 must belong to $base(S')$, hence $S' = S_1$ or $S' = \Omega$.

In other words, the role of the forward implication is to propagate an input transition. To do so, it must ensure that any state that can be reached after the input transition under consideration is computed, and the the set of enabling transitions is updated in consequence. The role of the backward implication is to propagate an output transition. To do so, it must ensure that any state that can be reached after the output transition under consideration is computed, and it may

remove from the set of enabled transitions the sequences that do not begin by the fired output transition. It is important to note that the definitions of the functions include the possibility of any overapproximation: in other words, any function that computes a bigger set than a given (forward, backward) implication is a (forward, backward) implication itself.

Now, some notions must be introduced in order to prove the correctness of the method. It is possible first to define a transition relation on $\mathcal{D}_{\mathcal{I}}^{\sharp}$ using the given forward and backward implication, as follows: if x is an input transition, then $X \Rightarrow_{FB}^x Y$ if and only if $Y = forw(x, X)$. If x is an output transition, then $X \Rightarrow_{FB}^x Y$ if and only if $Y = back(x, X)$. It makes it possible to define the *symbolic concretization function*, $conc^{\sharp} : \mathcal{D}_{\mathcal{S}} \rightarrow \mathcal{D}_{\mathcal{I}}^{\sharp}$, as

$$conc^{\sharp}(\phi) = \bigsqcup_{\exists w \in (I+O)^*, \phi_0 \xrightarrow{w} \phi, S_0 \Rightarrow_{FB}^w S} S,$$

where $S_0 \in \mathcal{D}_{\mathcal{I}}^{\sharp}$ satisfies $\alpha(q_0) \subseteq base(S_0)$. Then, the main result about the correctness of the method is the following result:

Theorem 1 For all $\phi \in \mathcal{D}_{\mathcal{S}}$, $base(conc^{\sharp}(\phi)) \supseteq conc(\phi)$.

In other words, all reachable implementation states have effectively been collected, that is exact checking needs to explore every implementation state that is reached through the execution of the system. The theorem states that, using backward and forward implication, the algorithm visits symbolic states which represent sets of implementation states that contain at least the exact set of reachable states.

In consequence, this framework can be used for conservative checking of safety properties. In other words, checking is performed on the computed $conc^{\sharp}(\phi)$ states, and the theorem states that these are actually larger than the real set of reachable states. Hence, if there is a safety failure in one of the reachable implementation states, it is successfully found, but since the framework is overapproximate, *false negatives* can appear (that is, detect safety failures that in fact do not occur), but no *false positive* (that is, if there is a safety failure, then it is detected).

In order to apply the proposed framework to a given problem, first, a symbolic representation should be defined, as well as a procedure to compute the least upper bound, and second, appropriate forward and backward implication functions should be defined. Then, Algorithm 2.1 can be used. It computes a mapping $conc^{\sharp}$ from $\mathcal{D}_{\mathcal{S}}$ to $\mathcal{D}_{\mathcal{I}}^{\sharp}$, which is kept in K , and associates exactly one symbolic state to each specification state.

Algorithm 2.1: procedure toplevel()

Output: K mapping $\phi \in \mathcal{D}_S$ to $K[\phi] \in \mathcal{D}_T^\#$.
foreach $\phi \in \mathcal{D}_S$ **do**
 | $K[\phi] := \text{undef}$;
endfch
to_visit := $\{\phi_0\}$;
 $K[\phi_0] := \text{compute initial sym. state}$;
while to_visit $\neq \emptyset$ **do**
 cur := **pop**(to_visit);
 foreach (x, dst) such that $S : \text{cur} \rightarrow^x \text{dst}$ **do**
 if x *input transition* **then**
 | $S := \text{forw}(x, K[\text{cur}])$;
 else
 | $S := \text{back}(x, K[\text{cur}])$;
 endif
 if $K[\text{dst}] = \text{undef}$ **then**
 | $K[\text{dst}] := S$;
 | **push**(dst, to_visit);
 else
 | $S' := S \sqcup K[\text{dst}]$;
 | **if** $S \neq S'$ **then**
 | $K[\text{dst}] := S'$;
 | **push**(dst, to_visit);
 | **endif**
 endif
 endfch
endw

3 Behaviour of timed circuits

3.1 Definitions

3.1.1 Using timing information in state machines

This paragraph explains how to represent timing information in the proposed framework. A state machine has an alphabet of transitions, and three particular sets are used : I, O, N . In the untimed case, such sets contain only transitions (that is, a transition might be a^+ to express a rising transition on wire a). In the timed case, this is extended with a non-negative real number $t \in \mathbb{R}_+$ as, for example, $(a^+, 2.4)$ which is a *timed event* representing a rising transition on wire a that occurs after 2.4 time units. The transition relation of a state machine is given as a set of elements (ϕ, x, ϕ') ; here x is (w^*, t_l, t_h) , where w is the name of a wire, $*$ is either $+$ or $-$ and t_l, t_h are non-negative real numbers, t_h being possibly $+\infty$. t_l and t_h are the bounds for the firing time of the transition. That is, if at time t , the system has moved to state ϕ , and transition $T = (\phi, (w^*, t_l, t_h), \phi')$ is the transition to be followed, then the next state of the system is ϕ' , and the change occurs at time t' where $t_l \leq t' - t \leq t_h$. Such timed state

machines are possibly infinite, but this paper restricts to the case where it is possible to define, for the specification state machine, a finite number of equivalence classes of timed states. There is no such restriction for the implementation state machine, since the proposed method does not need to handle it directly.

3.1.2 Timed asynchronous circuits

A *Timed Asynchronous Circuit* is given as a tuple (W_I, W_O, W_N, G) where W_I is the set of the *primary inputs*, W_O the *primary outputs*, W_N the *internal signals*, and $G : (W_O + W_N) \rightarrow \mathbb{R}_+ \times \mathbb{R}_+ \times \mathcal{F}$ is the *gate definitions*, where \mathcal{F} is the set of Boolean formulas over the variables in $W_I + W_O + W_N$, and $\mathbb{R}_+ = \mathbb{R}_+ \cup \{+\infty\}$.

If $w \in W_O + W_N$, then $G(w) = (l, u, b)$ where l is the *lower bound* of the gate, u the *upper bound* and b the *Boolean formula*. The semantics is as follows. When a change in the inputs of the gate at time t causes a change in the evaluation of the Boolean function b , the actual value of wire w changes at time t' , where $l \leq t' - t \leq u$.

The *implementation state machine* of a timed asynchronous circuit is a timed state machine, as discussed in the previous paragraph. Even though the actual computation of the implementation state machine from a given netlist is a complex task, it does not need to do it explicitly, as all relevant manipulations are done through the netlist.

For a signal w , $\text{fanin}(w)$ denotes its fanin and $\text{fanout}(w)$ its fanout in the current circuit.

3.1.3 Difference bound matrices

Let T be an infinite set of *timers*. Assume there is a special element $\bar{\kappa} \in T$, called the *current timer*. A *Difference Bound Matrix* (DBM) is a pair $D = (B, M)$ where B is a finite subset of T such that $\bar{\kappa} \in B$ and M is a two-dimensional vector $M \in \overline{\mathbb{R}}^{B \times B}$. B is the *set of timers* of the DBM, and M is its *underlying matrix*, where $\overline{\mathbb{R}}$ is $\mathbb{R} \cup \{+\infty\}$, and \mathbb{R} is the set of real numbers.

An *assignment* of a set B of timers is a vector in \mathbb{R}^B . The set of *solutions* of $D = (B, M)$ is the set of all assignments α of B , such that for all $\tau_1, \tau_2 \in B$, $\alpha(\tau_1) - \alpha(\tau_2) \leq M[\tau_1, \tau_2]$. The set of solutions of D is denoted by $\|D\|$. It is possible to consider the following ordering relation on a DBM: if $D = (B, M)$ and $D' = (B', M')$, then $D \leq D'$ if and only if $B = B'$ and $\forall \tau, \tau' \in B, M[\tau, \tau'] \leq M'[\tau, \tau']$. A DBM D is said to be *canonical* if and only if for all other DBM D' such that $\|D\| = \|D'\|$, $D \leq D'$. Canonicalization of a DBM is typically performed using the Floyd-Warshall algorithm.

3.2 Behaviour of the wires

This section gives a semantics to timed asynchronous circuits that satisfies the following (qualitative) properties:

- it is conservative;
- it is algorithmically easy to handle.

To do so, we focus on the behaviour of one particular wire in one particular state of the system: what transitions it can make, and what are the timing constraints associated with the transitions. Timing constraints are represented using DBMs.

3.2.1 Formalism

The set of *directions* is $\Delta = (\mathbf{1}, \mathbf{0}, [\uparrow], [\downarrow])$. $\mathbf{1}$ represents a wire which stays high, $\mathbf{0}$ a wire which remains low, $[\uparrow]$ a wire that makes a rising transition and $[\downarrow]$ a wire that makes a falling transition. We now demonstrate why additional elements in Δ are not needed. In a given state, the *behaviour* of wire w is given as a triple (δ, τ, τ_h) with $\delta \in \Delta$, timers $\tau, \tau_h \in T \cup \{\text{none}\}$. This is represented as $w : \delta \frac{\tau}{\tau_h}$. δ is the *direction* of the behaviour, τ is the *standard timer* and τ_h the *hazard timer*. τ is *none* if and only if δ is $\mathbf{1}$ or $\mathbf{0}$. τ_h is *none* if and only if there is no hazard on the wire.

τ is used to keep track of the timing constraints relative to the firing of a rising or falling transition on wire w . That is, for a DBM D containing the current relations between the timers in the system, if there exists $\alpha \in \llbracket D \rrbracket$ with $t = \alpha(\tau)$, and e.g. $\delta = [\uparrow]$, then the rising transition on w can take place at time t .

τ_h is used to keep track of the information concerning undefined behaviour on w . That is, if $\tau_h \neq \text{none}$ for some $\alpha \in \llbracket D \rrbracket$ with $t = \alpha(\tau_h)$, then the value of wire w at time t is undefined/unknown.

The set of behaviours is denoted by \mathbf{Bhv} .

3.2.2 Computing the behaviours of gates

Using the proposed formalism, the behaviour of the output of any gate can be determined (eventually in a conservative fashion) using only the behaviours of the inputs of this gate. In other words, a *syntax directed* algorithm can be used to compute the behaviour of any wire in a circuit starting with the (given) behaviours of its primary inputs.

This computation is separated into two (related) parts: the computation of the direction of the output wire w , and the updating of the DBM in order to include the information about the timers of the behaviour of this wire. The first part is straightforward, using a table of operations: for example, if $c = a \wedge b$ and a is $\mathbf{1}$, then c has the same direction as b ; the point is that if a is $[\uparrow]$ and b is $[\downarrow]$, the result is ultimately $\mathbf{0}$ but there is a transient period of time where the value of the

gate is undetermined. This is represented using the hazard timer of the behaviour.

The second part is less straightforward, as four operations must be introduced: the *timer union*, as well as the *timer minimum*, *maximum*, and the *global union*. The timer union, denoted by $M' = M[\tau'' := \tau \cup \tau']$, updates the DBM M into M' , by adding the new timer τ'' satisfying the following condition: for all $\alpha \in \llbracket M \rrbracket$, there exists $\alpha', \alpha'' \in \llbracket M' \rrbracket$ such that α, α' and α'' take the same value everywhere except on τ'' , and that $\alpha'(\tau'') = \alpha(\tau)$, and $\alpha''(\tau'') = \alpha(\tau')$. In other words, admissible values for τ'' are values admissible either for τ or τ' . Since the requirement is only the existence of such α', α'' , an implementation is free to use any kind of overapproximation.

Similarly, for the minimum: $M' = M[\tau'' := \min(\tau, \tau')]$, M is updated to M' , such that if $\alpha \in \llbracket M \rrbracket$, then there exists $\alpha' \in \llbracket M' \rrbracket$, identical to α everywhere but on τ'' , and $\alpha'(\tau'') = \min(\alpha(\tau), \alpha(\tau'))$. The maximum operation is defined similarly.

Finally, the global union of DBM M and M' , denoted by $M \hat{\cup} M'$, is a matrix M'' satisfying $\llbracket M \rrbracket \cup \llbracket M' \rrbracket \subseteq \llbracket M \hat{\cup} M' \rrbracket$.

Using these operations, it is possible to fully combine behaviours into a new behaviour. Let us consider the example of a delay-less *AND*-gate $c = \text{AND}(a, b)$. The rest of this paragraph shows two different examples:

- $a : [\uparrow] \frac{\tau_a}{\tau_a}$ and $b : [\uparrow] \frac{\tau_b}{\tau_b}$. Then c rises, and it does so as soon as the latest of a and b has done so. Thus, $c : [\uparrow] \frac{\tau_c}{\tau_c}$, where the standard timer τ_c is obtained as the maximum of τ_a and τ_b , and τ_c^h is the union of the timers τ_a^h and τ_b^h , which is actually an approximation of the exact regroupment of behaviours. In other words, the updated DBM is

$$D' := D[\tau_c := \max(\tau_a, \tau_b); \tau_c^h := \tau_a^h \cup \tau_b^h]$$

- $a : [\uparrow] \frac{\tau_a}{\text{none}}$ and $b : [\downarrow] \frac{\tau_b}{\text{none}}$. Then, depending on the relative orderings of τ_a and τ_b , either $c : \mathbf{0} \frac{\text{none}}{\text{none}}$ (if the rising transition always take place after the falling transition), or $c : \mathbf{0} \frac{\text{none}}{\tau_c^h}$ (if it is possible for a to rise before b has fallen). In the first case, no modification to the DBM is necessary; in the latter case, the DBM is updated to reflect that a hazard is possible for $\tau_a \leq \tau_c^h \leq \tau_b$.

3.2.3 Handling complex gates

It is well known that any complex gate can be represented using only *AND* gates and *NOT* gates. In the case of timed circuits, general timed gates can be represented by a delay-less complex gate, followed by a single timed buffer. The

only exception concerns memory elements, the representation of which must be taken special care of. The manipulations referred to in this subsection only concern the delay-less part of the timed gates, thus they preserve the semantics of the circuits.

Following the proposed methodology, it is only needed to define how to combine behaviours through these three kinds of gates in order to be able to handle any complex gate. The delay-less *NOT* gate is simple, as its only effect is to inverse the direction of the behaviour, and the delay-less *AND* gate has been described previously. Finally, the delayed buffer is simple, too, since its only effect is to add some delay to the hazard timer and the standard timer, and that relation can be directly expressed using the DBM. Hence, the full problem of obtaining the behaviour of gates in a circuit has been reduced to the problem of computing the operation table of three different gates. Then, these three operation tables can be computed once and for all (typically, by hand; the process is tedious but simple) and an efficient way to compute the behaviour of internal and output wires in a circuit is obtained, for any complex gate.

4 Hazard checking

Hazard checking is the analysis of the behaviour of a given design with respect to a given specification, in order to verify that no improper sequence of operations due to internal delays can cause a safety failure. In addition to be a critical part of correctness, it is also a central problem during technology mapping of asynchronous circuits. More formally, the input of a hazard checking algorithm is both a specification (given as a timed state machine or a time petri net) and an implementation (given as a timed asynchronous circuit), and the output is a list of couples (w, ϕ) where gate w has a hazard in specification state ϕ .

The next subsection defines the symbolic representation for this problem, and the three following subsections show the relevant algorithms.

4.1 Applying the general framework

In order to apply the general framework to the problem at hand, the relevant data must be defined.

\mathcal{D}_S is the set of states of the specification, given as a finite timed state machine. $\mathcal{D}_{\mathcal{I}}^{\#}$ is the set of all pairs (D, v) where D is a DBM and v is a vector that maps a wire to a behaviour, that is, $v \in \mathbf{Bhv}^{W_I+W_O+W_N}$. D contains the relative timer orderings and v contains the particular behaviour of each wire.

From any symbolic state $S = (D, v)$, it is possible to derive a set of sequences of transitions. For example, if $v(a) = [\uparrow]_{\frac{\tau_a}{none}}$ and $v(b) = [\downarrow]_{\frac{\tau_b}{none}}$, and $D : 2 \leq \tau_b - \bar{\kappa} \leq 4, 1 \leq \tau_a - \bar{\kappa} \leq 3$, then there is a solution α such that

$\alpha(\tau_a) = 1, \alpha(\tau_b) = 2$, and the corresponding sequence of transitions would be: $[(a^+, 1); (b^-, 1)]$. If there are hazard timers (that is, their value is not *none*), then they represent that the corresponding wire is able to make any transition at any time specified by that hazard timer.

Hence, it is possible to define \mathbb{T} as the set of sequences of timed events that satisfy (D, v) . $base(S)$ is then the following set:

$$base(S) = \{q \in \mathcal{D}_{\mathcal{I}}, post^{\mathbb{T}}(q) \neq \emptyset\}$$

In this application, $\mathcal{D}_{\mathcal{I}}^{\#}$ is chosen with great care. Thus, the definition of $base$ is straightforward. All algorithms in this section are determined by this function. However, once the algorithms have been derived, it is not necessary to handle the $base$ function directly anymore.

4.2 Least upper bound of behaviours

This subsection explains the algorithms for computing the least upper bound using a simple example. Let us consider the following behaviours: $S_1 = (D_1, v_1)$ and $S_2 = (D_2, v_2)$ where $D_1 = 2 \leq \tau - \bar{\kappa} \leq 4, D_2 = 3 \leq \tau - \bar{\kappa} \leq 5, v_1[a] = [\uparrow]_{\frac{\tau}{none}}, v_2[a] = [\uparrow]_{\frac{\tau}{none}}$. In this example, in S_1 , a is enabled to rise after between 2 and 4 time units, and in S_2 , it is enabled to rise after between 3 and 5 time units. The least upper bound of S_1 and S_2 is computed as follows. It must represent any sequence of transitions possible from S_1 and S_2 , and be minimal in this respect. Then, S must be $S = (D, v)$ where $D = 2 \leq \tau - \bar{\kappa} \leq 5$ and $v[a] = [\uparrow]_{\frac{\tau}{none}}$, as a may rise after between 2 and 5 time units in either S_1 or S_2 . Such D is computed by using the global union: $D = D_1 \hat{\cup} D_2$. The reader may check that $base(S) \supseteq base(S_1) \cup base(S_2)$. In the general case, for each wire, depending on its direction in S_1 and S_2 , its direction and timers are determined in the resulting symbolic state S using a table (as shown in Figure 4). The table shows, for each pair of directions, what is the direction of the wire in the least upper bound. The computation of the standard timer is split into 4 cases (*a, b, c.1* and *c.2*).

- case *a*: the direction in S_1 is equal to the direction in S_2 . As shown in the example at the beginning of this subsection, the resulting direction is the same as the direction in S_1 and S_2 and the resulting DBM is the global union of the DBMs of S_1 and S_2 .
- case *b*: the direction in S_1 and in S_2 are opposite (**0** and **1**, or $[\uparrow]$ and $[\downarrow]$). Let us take the example where the direction in S_1 is **0** and the direction in S_2 is **1**. Then, in the least upper bound of S_1 and S_2 , at any time, the value of the wire can be either 0 (since it is low in S_1) or 1 (since it is high in S_2). As a result, any possible behaviour of this wire is possible. This is

	0	1	\uparrow	\downarrow
0	0 (<i>a</i>)	X (<i>b</i>)	0 (<i>c.1.</i>)	0 (<i>c.2.</i>)
1	X (<i>b</i>)	1 (<i>a</i>)	1 (<i>c.2.</i>)	1 (<i>c.1.</i>)
\uparrow	0 (<i>c.1.</i>)	1 (<i>c.2.</i>)	\uparrow (<i>a</i>)	X (<i>b</i>)
\downarrow	0 (<i>c.2.</i>)	1 (<i>c.1.</i>)	X (<i>b</i>)	\downarrow (<i>a</i>)

Figure 4. Least upper bound computation.

represented by setting the direction in S as any direction, and the hazard timer as τ_h , a new timer with no constraint (that is, at any moment, the value of the wire is undetermined). The case \uparrow , \downarrow is treated similarly. Figure 4 uses an **X** to represent this behaviour.

- case *c.1.*: for example, the direction in S_1 is **0** and the direction in S_2 is \uparrow . Then, before the rising transition in S_2 takes place, in both S_1 and S_2 , the value of the wire is **0**, and after the transition takes place, the value becomes undetermined (**0** in S_1 and **1** in S_2). This is represented by setting the direction of the result to τ_h a new timer, with the constraint $0 \leq \tau_h - \tau_2 \leq +\infty$, where τ_2 is the standard timer of the rising transition in S_2 .
- case *c.2.*: this situation is symmetrical to the situation in *c.1.*

4.3 Forward implication in \Rightarrow_{FB}

The forward implication, as defined in Section 2, is defined as the operator that “reads” an input transition² and updates the state of the internal wires accordingly. This is exactly what the recomputation of the behaviours does: from a change of an input signal, *behaviour combination* (see Section 3.2.3) can be used to obtain the updated behaviours of the internal signals. This is shown in Algorithm 4.1.

First, a *maximal number of visits*, MAX , is chosen (typically, 2 or 3). Then, the netlist is explored, from the modified input, always visiting outputs of a gate after having visited this gate. For example, let us suppose the following definition: $b = \neg a$, $c = b \wedge c$. Then, if a is modified, b , c are visited, and then c again. When visiting a node for the first time, the current symbolic state is stored. On the next visit, the algorithm compares the newly computed symbolic state with the stored one for this wire. If they are equal, then the algorithm need not propagate from this gate anymore, because a fixed point in the evaluation has been reached. If not, then the algorithm continues exploration, replacing the current symbolic state by its least upper bound with the stored one, until this gate has been visited

²When there is feedback, output transitions are input transitions too.

Algorithm 4.1: procedure $forw(w_0, S)$

Input: w_0 the primary input which value has changed
Output: the updated symbolic state of the system

```

foreach  $w$  wire do
  | visits[ $w$ ] := 0;
endfch
( $D, v$ ) :=  $S$ ;
 $v[w_0]$  := the new behaviour of  $w_0$  w.r.t.  $S$ , updating  $D$ ;
to_visit := { $w_0$ };
while to_visit  $\neq \emptyset$  do
  |  $cur$  := pop(to_visit);
  | if visits[ $cur$ ] <  $MAX$  then
    |  $v[cur]$  := the new behaviour of  $cur$  w.r.t.
    | ( $D, v$ ), updating  $D$ ;
    | if visits[ $cur$ ] = 0 then
      |  $ss[cur]$  := ( $D, v$ );
      | to_visit := to_visit  $\cup$  fanout( $cur$ );
    | else
      | ( $D, v$ ) := lub(( $D, v$ ),  $ss[cur]$ );
      | if ( $D, v$ )  $\neq$   $ss[cur]$  then
        | to_visit := to_visit  $\cup$  fanout( $cur$ );
        |  $ss[cur]$  := ( $D, v$ );
      | endif
    | endif
  | else
    | remove constraints concerning the standard
    | timer and the hazard timer of  $v[cur]$  in  $D$ ;
    | report a possibly unstabilized wire;
  | endif
  | visits[ $cur$ ] := visits[ $cur$ ] + 1;
endw
return ( $D, v$ );

```

at least MAX times. If no stabilization in the behaviour has occurred, the algorithm has detected that this gate is unstable, and its behaviour is set to the fully unknown behaviour. Thus, an unconstrained hazard timer is used for this wire. If MAX is set to a value of 2 or 3, then experimentally, conservativeness occurs only very rarely.

The algorithm is clearly terminating, as each wire is visited at most MAX times. As DBM operations have a cubic complexity, propagation itself has a complexity of $O((|I| + |O| + |N|)^3 \times (|O| + |N|))$, that can be considered as a biquadratic (n^4) complexity in the number of wires of the implementation.

4.4 Backward implication in \Rightarrow_{FB}

Backward implication, shown in Algorithm 4.2, is called when an output is fired. In this case, it only needs to check, for every timer τ such that, if D is the current DBM, then $D : \tau \leq \tau_0$ where τ_0 is the standard timer of the output

Algorithm 4.2: procedure *back*(S, w_0)

Input: w_0 the primary output which has changed**Output:** the updated symbolic state of the system

```
 $\delta_0 \frac{\tau_0}{none} := v[w_0];$ 
foreach  $w$  wire do
   $\delta \frac{\tau}{\tau_h} := v[w];$ 
  if  $D[\tau, \tau_0] \leq 0$  then
    REM:  $w$  is known to have fired;
    if  $\delta = [\uparrow]$  then
       $v[w] := \mathbf{1} \frac{none}{\tau_h};$ 
    else
      REM:  $\delta = [\downarrow];$ 
       $v[w] := \mathbf{0} \frac{none}{\tau_h};$ 
    endif
    project out  $\tau$  from  $D$ ;
  endif
  if  $D[\tau_h, \tau_0] \leq 0$  then
     $v[w] := \delta \frac{\tau}{none};$ 
    project out  $\tau_h$  from  $D$ ;
  endif
endfch
return ( $D, v$ );
```

transition that just fired and τ is the standard timer of some wire $w : \delta \frac{\tau}{\tau_h}$. In this case, it is known that the transition on w has fired, thus, δ is set to its stable value ($\mathbf{1}$ for $[\uparrow]$, $\mathbf{0}$ for $[\downarrow]$), and τ is projected out of the current DBM. If, for some τ_h , $D : \tau_h \leq \tau_0$ where there is a wire $w : \delta \frac{\tau}{\tau_h}$, then this hazard transition has caused possible hazards in the past, but they have no direct effect on the future behaviour of the circuit. Hence, the behaviour of w is set to $\delta \frac{\tau}{none}$ and τ_h is projected out.

The *back* procedure does not handle the case where there is a hazard timer on wire w_0 . The reason is that, during the main procedure, when a hazard is detected on a primary output such as w_0 , it is reported, then removed before the call to the *back* procedure.

4.5 Detection of hazards

Failure states are symbolic states for which there exists at least one primary output with an existing hazard timer. Unlike previous approaches [1, 7], a hazard on an internal wire is not a case of failure when it can be proved to be absorbed later. As shown in Algorithm 2.1, hazards are detected just after the call to *forward*, and the pair (x, cur) is added to the list of hazards.

Table 1. Experimental results (timed)

Example	#G	#h	#d	#at	new	atacs
alloc-outbound	11	0	0	0	0	0
chu133	9	1	1	1	0	0
ebergen	9	3	3	3	0	0
half	7	1	1	1	0	0
sbuf-send-pkt2	13	0	0	1	0.13	0.05
sbuf-ram-write	17	1	1	2	0.04	0.06
rpdft	8	1	1	3	0.035	0.029
vme	12	1	1	–	0.05	–
dff	6	2	2	–	0.08	–
trimos-send	24	5	5	5	2.43	1.7

execution times are given in seconds

#G is the number of gates, #h is the exact number of hazards, #d is the number of hazards detected by the proposed method, #at is the number of hazards detected by ATACS, new is the run time of the proposed method, ATACS is the run time of ATACS. 0 represents a run time inferior to 10ms. Entries labelled as – indicate that ATACS was unable to check the given circuit due to internal loops.

5 Experimental results

Experimental results shown in Table 1 were obtained on a Pentium IV 2GHz processor, with 3GB of RAM. The proposed approach has been compared to the implementation of [7] in the ATACS tool.

We also applied our framework to the checking of speed-independent circuits, using different adequate semantics for the behaviour of wires, obtaining the results in Table 2, in which a comparison of the algorithm using our new framework to an implementation of [1] in ATACS is given, as well as to versify [9], that uses a symbolic representation of states to reduce the cost of full exploration. The examples used are hazard-free, and the three algorithms detect this correctly.

The experimental results show that our algorithm has a speed that compares well with [7], which is currently the fastest algorithm for the hazard checking of timed circuits, as both of them do not explore the full state space by using different representations for internal wires.

Furthermore, while the algorithm is conservative, false negatives have not been encountered, and it is possible to give the correct number of hazards for examples for which [7] could either not check because of the presence of internal loops (as in examples vme and dff), or gave false negatives (as in examples sbuf-send-pkt2, sbuf-ram-write and rpdft).

6 Conclusion

This paper first presents a new general framework for the conservative exploration of the state-space of an implementation with respect to a given specification. In order to apply this framework to the problem of hazard-checking of

Table 2. Experimental results (untimed)

Circuit	#G	new	atacs	vers.
and_6	128	0	0.7	5.4
and_8	512	0.7	5.7	618.3
simple	6	0	—	0.01
trimos_s	15	0.16	0.07	0.72
master_r	19	4.6	3.6	7.96
wrdatab	20	0.25	0.078	1.01
IIR_2mul_2	22	0.15	0.1	1.65
FIR5_2mul	43	45.7	—	214.8

execution times are given in seconds

0 represents an execution time inferior to 10ms.

— : the examples could not be checked by ATACS due to internal loops.

timed asynchronous circuits, this paper next shows a new approach to the representation of the behaviour of such circuits, based on an abstracted representation of the links between enabled transitions in a given specification state. The resulting method is a fast, conservative yet precise algorithm for the hazard-checking of timed asynchronous circuits.

This algorithm can potentially be used to eliminate part of the detected hazards by using a simple analysis of the timing relations and inserting adequate delay buffers. While we believe such correction is sufficient to eliminate most of the detected hazards, we plan to enhance the algorithm used to remove hazards, in order to be able to derive a fully hazard-free design from a hazardous netlist given as an input.

References

- [1] P. A. Beerel, T. H.-Y. Meng, and J. Burch. Efficient verification of determinate speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 261–267. IEEE Computer Society Press, Nov. 1993.
- [2] W. Belluomini, C. J. Myers, and H. P. Hofstee. Timed Circuit Verification Using TEL Structures. *IEEE Transactions on Computer-Aided Design*, 20(1):129–146, Jan. 2001.
- [3] R. Clarisó and J. Cortadella. Verification of concurrent systems with parametric delays using octahedra. In *Proc. 5th International Conference on Application of Concurrency to System Design (ACSD'05)*. IEEE Computer Society Press, June 2005.
- [4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
- [5] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, Apr. 1959.
- [6] C. J. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.
- [7] C. Nelson, C. Myers, and T. Yoneda. Efficient verification of hazard-freedom in gate-level timed asynchronous circuits. *IEEE Transactions on CAD*, page 26(3), 2007.
- [8] M. A. Peña, J. Cortadella, A. Kondratyev, and E. Pastor. Formal verification of safety properties in timed circuits. In *Proc. International Symposium on Advanced Research*

in Asynchronous Circuits and Systems, pages 2–11. IEEE Computer Society Press, Apr. 2000.

- [9] O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *16th International Conference on the Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 374–391, June 1995.
- [10] T. Rokicki and C. Myers. Automatic verification of timed circuits. *LNCS 818 Computer Aided Verification*, pages 468–480, 1994.
- [11] K. S. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. J. Myers, K. Y. Yun, R. Koi, C. Dike, and M. Roncken. An asynchronous instruction length decoder. *IEEE Journal of Solid-State Circuits*, 36(2):217–228, Feb. 2001.
- [12] I. Sutherland and S. Fairbanks. GasP: A Minimal FIFO Control. *Proc. of ASYNC'01*, pages 46–53, 2001.
- [13] S. Thompson and A. Mycroft. Abstract interpretation of combinational asynchronous circuits. In R. Giacobazzi, editor, *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 2004.
- [14] T. Yoneda and H. Ryu. Timed trace theoretic verification using partial order reduction. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 108–121, Apr. 1999.
- [15] H. Zheng, E. Mercer, and C. J. Myers. Modular verification of timed circuits using automatic abstraction. *IEEE Transactions on Computer-Aided Design*, 22(9), Sept. 2003.