Lazy transition systems: application to timing optimization of asynchronous circuits *

Jordi Cortadella, Univ. Politècnica de Catalunya, Barcelona, Spain
Michael Kishinevsky, Intel Corp., Hillsboro, Oregon
Alex Kondratyev, The University of Aizu, Japan
Luciano Lavagno, Politecnico di Torino, Italy
Alexander Taubin, The University of Aizu, Japan
Alex Yakovlev, University of Newcastle upon Tyne, United Kingdom

Abstract

This paper introduces *Lazy Transitions Systems* (LzTSs). The notion of laziness explicitly distinguishes between the enabling and the firing of an event in a transition system.

LzTSs can be effectively used to model the behavior of asynchronous circuits in which relative timing assumptions can be made on the occurrence of events. These assumptions can be derived from the information known a priori about the delay of the environment and the timing characteristics of the gates that will implement the circuit. The paper presents necessary conditions to synthesize circuits with a correct behavior under the given timing assumptions.

Preliminary results show that significant area and performance improvements can be obtained by exploiting the extra "don't care" space implicitly provided by the laziness of the events.

1 Introduction

In the recent years, there has been significant progress in developing methods and tools for asynchronous circuit synthesis [21, 17, 19, 25, 5]. The two chief directions in this work have been following the two, traditionally competing, synthesis approaches, one based on the Huffman's state machine model [7, 22], the other deriving from Muller's concept of a *speed-independent* circuit [14]. The former, also known as *fundamental mode* circuit design, makes strong assumptions about the delay of the environment compared to that of the circuit. It requires that the environment be slow enough in applying the new input values so as to allow the circuit to stabilize after responding to the previous input. The most well-known method associated with this approach is the one called *Burst-Mode* (BM) circuit design, developed in [19, 26]. The second approach, on the contrary, makes no assumptions about the delays in the environment, permitting it to switch some of the inputs in response to changes in some of the circuit's outputs, without waiting for their complete stabilization. This way of action is often called *input-output* (IO) mode. To define the behavior of the circuit and the environment interacting in the IO mode, one normally uses an event-based description rather than a state-oriented one like in the BM approach. The recently developed design methods and software based on Signal Transition Graphs (STGs) [10, 5] exemplify this approach, and produce speed-independent circuits, whose behavior is invariant to delays in gates but may be sensitive to wire delays.

Although the second approach looks more flexible on the surface than the BM one, and promises higher performance and modularity, in reality it does not always come as efficient as expected, both in speed and area concerns. This is especially true with the advent of deep-submicron technologies, which radically change the ratio between gate and wire delays. While being conservative to the former it is overly optimistic to the latter. Even though the notion of (extended) isochronic forks [11, 23] can help in guiding the technology mapping of speed-independent circuits towards safer solutions, it does not resolve the fundamental problem of time dependence for wires. On the other hand, in order to guarantee correct action regardless of the delays of both circuit gates and the environment, the synthesis process often caters for potential concurrency which will not exist in reality. This often results in excessively redundant implementations, which lose to their possible BM counterparts both in speed and area.

In order to battle the problems characteristic to both of the above mentioned "extremes" a method, called *timed circuits*, has been developed in [18, 17]. The main idea of this method is to retain the flexibility of an event-based IO approach but make the circuit's implementation more realistic and therefore more efficient. The awareness of time, required for asynchronous controllers to be on a par with synchronous ones in speed [6], is achieved by associating explicit timing information with the actions performed by the environment and by the circuit, and utilizing it throughout the design procedure to optimize the final logic. The major effect of timing assumptions applied to circuit design is following. Such constraints can reduce the state space effectively reachable by the circuit. Hence, firstly, they can 'eliminate' some undesirable states, e.g. where input events might disable some output signal transitions, or 'resolve' state coding problems, e.g. the presence of semantically different states with identical codes. Secondly, they can help optimize logic by exploiting the additional "don't care" space. Thirdly, the timing information may assist in allowing some, relatively slow, actions to be started earlier than they would normally be allowed in the speed-independent case; their actual firing with respect to other events will remain unchanged. Finally, such timing information can be made global enough to cover the BM designs; indeed, it can be shown (cf. Section 4) that a BM design is just a special case of a timed circuit with *simultaneity constraints*, wherein all outputs are assumed to change at once before any new input transition arrives. While work of Myers et al. [18, 17] appears to be exploiting the first two of the above-mentioned factors, it has not been able to provide an adequate formal support for the latter two issues.

In this paper, we target all these potential gains, by developing a behavioral model (Section 4) of a timed circuit, called *Lazy Transition System*. The two crucial novel elements of this model are:

- the concept of relative timing constraints, where the exact

* This work has been funded by ESPRIT ACiD-WG Nr. 214949, CICYT TIC 95-0419, EPSRC grants GR/L24038 and GR/K70175, Spain-UK Acciones Integradas Programme 1998/99, and MURST (project "VLSI Architectures").

timing information is abstracted away. Instead, we use only difference (event a fires earlier than event b) or simultaneity (events a and b fire at the same time with respect to event c) assumptions. Such conditions can either be provided by the designer (which is the case at present) or produced by a hypothetical timing analysis tool [1].

- the notion of laziness that explicitly distinguishes between the enabling and the firing of an event in a transition system. This allows us not only exploit delays in reducing concurrency to simplify designs on the basis of a priori timing conditions but also to increase concurrency using the (backward) expansion of the set of enabling states. In the latter case, we also expect the designer to be able to trade off between speed and area increase.

The paper presents necessary conditions (Section 4) to synthesize circuits with a correct behavior under given timing assumptions and develops an algorithm (Section 5), implemented within the synthesis tool `petrify`. The preliminary experiments (Section 6) show significant area and performance improvements due to exploiting the extra "don't care" space implicitly provided by the laziness of the events.

2 Basic notions

In this section we present basic definitions that will be used in the paper. For brevity, we assume the reader to be familiar with Petri nets, a formalism used to specify concurrent systems. We refer to [15] for a general tutorial on Petri nets.

2.1 State Graphs

A *State Graph* (SG) is a labeled directed graph whose nodes are called *states*. Each arc of an SG is labeled with an *event*, that is a rising ($a+$) or falling ($a-$) transition of a signal a in the specified circuit. We also allow the notation $a*$ if we are not specific about the direction of the signal transition. The set of signals of an SG is called $X = I \cup O$, where I and O denote the set of input and output signals respectively. The behavior of the input signals is determined by the environment whereas the behavior of the output signals must be implemented by the circuit. We write $s \xrightarrow{a} (s \xrightarrow{a} s')$ if there is an arc from state s (to state s') labeled with a.

A labeling function $v : S \rightarrow \{0,1\}^n$ assigns a vector of signal values to each state ($n = |X|$). We will call $v_a(s)$ the value of signal a in state s. An SG is *consistent* if:

$$s \overset{a+}{\leftrightarrow} s' \implies v_a(s) = 0 \wedge v_a(s') = 1$$

$$s \overset{a-}{\leftrightarrow} s' \implies v_a(s) = 1 \wedge v_a(s') = 0$$

$$s \overset{b*}{\leftrightarrow} s' \wedge a \neq b \implies v_a(s) = v_a(s')$$

2.2 Signal Transition Graph

A Signal Transition Graph (STG) is a Petri net in which transitions are labeled with the same type of events we have defined for SGs, i.e. rising and falling signal transitions [10].

An STG has an associated SG in which each reachable marking corresponds to a state and each transition between a pair of markings to an arc labeled with the same event of the transition.

Although STGs with bounded reachability space and SGs have the same descriptive power, STGs can usually express the same behavior more succinctly. In this paper, STGs help to illustrate timing assumptions in a more intuitive way.

Figure 1.a depicts an STG with three signals. For simplicity, places with only one input and output transitions are omitted. Figure 1.b shows the corresponding SG with states labeled with the binary vectors of the signal values. The SG is consistent.

[1] We believe that the level of research in this area, although being quite significant recently [3, 8, 16] is still insufficient to warrant practicality, especially in providing adequate relative timing for realistic circuit designs.

2.3 Properties for implementability

Further to consistency, the following two properties are necessary for an SG to be implemented by a speed-independent circuit [9].

The first property is *speed-independence*. It consists of three parts: determinism, commutativity and output-persistence. An SG is called *deterministic* if for each state s and each label a there can be at most one state s' such that $s \xrightarrow{a} s'$. An SG is called *commutative* if whenever two transitions can be executed from some state in any order, then their execution always leads to the same state, regardless of the order. An event a^* is called *persistent* in state s if it is enabled in s and remains enabled in any other state reachable from s by firing another event b^*. An SG is called *output-persistent* if its output signal events are persistent in all states and no output signal event can disable input events. In Figure 1.b, event $y+$ is persistent in the state 100, since the firing of $z+$ leads to the state 101 in which $y+$ is still enabled.

The second property, *Complete State Coding* (CSC), is necessary and sufficient for the existence of a logic circuit implementation. A consistent SG satisfies the CSC property if for every pair of states (s, s') such that $v(s) = v(s')$, the set of output events enabled in both states is the same.

The SG of Figure 1.b fulfils all the above properties.

2.4 Excitation and quiescent regions. Next-state function.

The *excitation region* of an event a^*, denoted by $\mathsf{ER}(a^*)$, is the set of states such that $s \in \mathsf{ER}(a^*) \Leftrightarrow s \xrightarrow{a^*}$. The *quiescent region* of $a+$, denoted by $\mathsf{QR}(a+)$, is the set of states such that $s \in \mathsf{QR}(a+) \Leftrightarrow v_a(s) = 1 \wedge s \notin \mathsf{ER}(a\leftrightarrow)$. Similarly, $s \in \mathsf{QR}(a\leftrightarrow) \Leftrightarrow v_a(s) = 0 \wedge s \notin \mathsf{ER}(a+)$.

In Figure 1.b, $\mathsf{ER}(x\leftrightarrow) = \{101, 111\}$ and $\mathsf{QR}(x\leftrightarrow) = \{001, 011, 010\}$. The symbol 0^* (1^*) indicates that a rising (falling) transition of the corresponding signal is enabled in that state.

The implementation of an SG as a logic circuit is done through the definition of the *next-state function* for each output signal and binary vector. It is defined as follows:

$$f_a(z) = \begin{cases} 1 & \text{if } \exists s \in \mathsf{ER}(a+) \cup \mathsf{QR}(a+) \text{ s.t. } v(s) = z \\ 0 & \text{if } \exists s \in \mathsf{ER}(a\leftrightarrow) \cup \mathsf{QR}(a\leftrightarrow) \text{ s.t. } v(s) = z \\ \leftrightarrow & \text{otherwise} \end{cases}$$

The next-state function f_a is correctly defined when the SG has the CSC property, i.e. when there is no pair of states (s, s') such that $v(s) = v(s')$ and $s \in \mathsf{ER}(a+) \cup \mathsf{QR}(a+)$ and $s' \in \mathsf{ER}(a\leftrightarrow) \cup \mathsf{QR}(a\leftrightarrow)$. Note that f_a is an incompletely defined function with a *don't care* (DC) set corresponding to those binary vectors without any associated state in the SG.

In the SG of Figure 1.b, the DC set is empty since all binary vectors have a corresponding state in the SG. As an example, $f(101) = 011$ since signals x and y are enabled in that state. The Karnaugh maps for the next-state functions are depicted in Figure 1.c.

2.5 Logic synthesis

From the next-state functions, a speed-independent circuit can be derived by implementing the boolean equation of each output signal as an atomic complex gate [14], as shown in Figure 1.d.

In general, the boolean equations may be too complex to be implemented as an atomic gate in a specific technology. Methods for logic decomposition and technology mapping that overcome this limitation have been proposed recently (e.g. [2, 4]). In this paper we do not address the problem of technology mapping. However, the optimization methods we propose can be easily combined with existing methods for logic decomposition that can be targeted to technology mapping into given gate libraries.

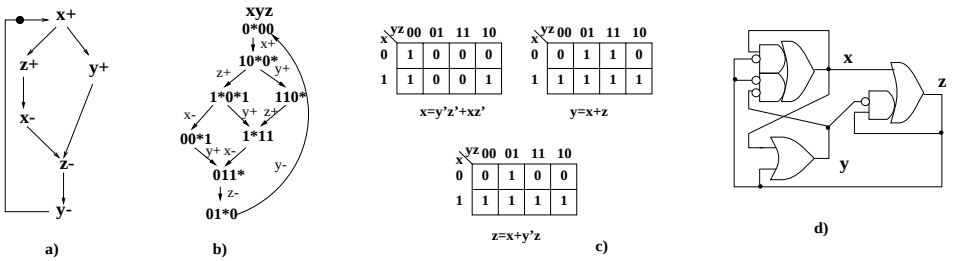

Figure 1: (a) STG, (b) SG, (c) next-state functions, (d) complex-gate implementation.

2.6 Monotonic covers

The following definition is related to hazards in the behavior of asynchronous circuits. It will be used later in the paper.

Given two sets of states S_1 and S_2 of an SG such that $S_2 \subseteq S_1$, and a transition $s \stackrel{a}{\rightarrow} s'$, we will say that S_1 is a *monotonic cover* of S_2 iff:

$$(s \in S_1 \setminus S_2 \implies s' \in S_1) \wedge (s \in S_2 \implies s' \notin S_1 \setminus S_2)$$

In the SG of Figure 1.b, the set $\{101, 110, 111\}$ is a monotonic cover of $\mathsf{ER}(x\Leftrightarrow)$. However, the set $\{100, 101, 111\}$ is not, since the transition 100 $\stackrel{y+}{\Leftrightarrow}$ 110 violates the conditions for monotonicity.

3 Motivating example

This section gives an intuitive picture of the optimizations based on timing assumptions. It is illustrated by an implementation of the *xyz* specification shown in Figure 1,a. A starting point for optimizations is given by a speed-independent implementation of *xyz* STG (see Figure 1,d).

Speed-independence gives a rather conservative view on gate delays: they are finite but arbitrary. However, when the gates of a circuit are adjacent on a chip (which is most likely for the modular style of implementation) one can expect from their delays to be related. This relationship might be expressed by matching the time for a signal propagation through different stages of logic. For example, one can assume that a signal propagates through a single gate faster than through k gates, where k is a technology and/or implementation dependent parameter.[2]

Let us assume that in a circuit for the *xyz* example two gate delays are always greater than a delay of a single gate. Under this assumption, even though the transitions $y+$ and $x\Leftrightarrow$are potentially concurrent in the SG, in an implementation $y+$ would always occur $before\ x\Leftrightarrow$. This timing assumption can be expressed in the SG by a special "timing arc" going from $y+$ to $x\Leftrightarrow$[24] (denoted in Figure 2,a by a dashed line). Timing restricts possible behaviors of implementation, in particular state 001 becomes *unreachable* because it can be entered only when $x\Leftrightarrow$fires earlier than $y+$. At unreachable states logic functions of output signals can be defined arbitrarily. Therefore use of timing assumptions increases the "don't care" space for circuit gates, which gives extra room for optimization.

For *xyz* example, putting 001 in the don't care set of z simplifies its function from $z = x + \overline{y}z$ to a buffer $z = x$ (see Figure 2,c,d).

To get more aggressive optimizations let us consider concurrent transitions $z+$ and $y+$ closer. These transitions are triggered by the same event $x+$ and due to the timing assumption $2 * gate_{min} > gate_{max}$ no gate can fire until both outputs y and z are set to 1. Therefore for all other signals of the circuit the difference in firing times of $y+$ and $z+$ is negligible. The latter means that for the rest of the circuit transitions $y+$ and $z+$ are *simultaneous and indistinguishable* and they can replace each other in causal relations with the other events.

In *xyz* example $x\Leftrightarrow$ is the only transition that can "hear" $z+$ or $y+$. The dashed hyper-arc from $z+, y+$ to $x\Leftrightarrow$in Figure 3,a graphically represents the simultaneity of $y+$ and $z+$ with respect to $x\Leftrightarrow$. Formally it means that for an enabling of $x\Leftrightarrow$we can choose any of the following conditions: 1) $z+$ 2) $y+$ 3) $z+ \vee y+$. This gives a set of states in which $x\Leftrightarrow$can be potentially enabled, i.e. the so-called *potentially enabling region* of $x\Leftrightarrow$ ($PEnR(x\Leftrightarrow)$) which is shadowed in Figure 3,b.

It is important to note that:

1) Even though $x\Leftrightarrow$ might be enabled in any state of $PEnR(x\Leftrightarrow)$ its firing (due to timing assumptions) can occur only when reaching state 111. This behavior will be called a *lazy* one, because after its enabling a signal is not eager to fire immediately but waits until certain states are entered.

2) A potentially enabling region gives an upper bound for the set of states in which a signal might be enabled. For a "real" enabling in an implementation we can choose a subset of the potentially enabling region. Playing with different sets of "real" enablings within a $PEnR$ gives new opportunities for the optimization of circuits.

Manipulations of signal enablings can be formalized by additional *don't cares* in the definition of a signal function. For the enabling of $x\Leftrightarrow$in xyz example a subset of $PEnR(x\Leftrightarrow) = \{101, 110, 111\}$ might be chosen. Transition $x\Leftrightarrow$fires at state 111 and therefore $x\Leftrightarrow$should be enabled in 111. Enabling of $x\Leftrightarrow$in the other two states 101 and 110 can be chosen arbitrarily, i.e. these states can be put in the don't care set of a function for x (see Figure 3),c. During minimization the function for x (which becomes simply an inversion) is defined to be 0 in state 110 and 1 in 101, i.e. minimization puts 110 into the set of enabled states of $x\Leftrightarrow$, while 101 is put into the set of states in which x is stable. Back-annotating this result to the level of event interaction gives an STG in Figure 3,e in which $x\Leftrightarrow$is triggered by $y+$ instead of causal relation $z+ \rightarrow x\Leftrightarrow$ in the original STG. This change of causal dependencies is valid under the assumption that $y+$ and $z+$ are simultaneous with respect to $x\Leftrightarrow$.

The timed circuit in Figure 3,d is much simpler than the speed-independent one in Figure 1,d. Nevertheless if the timing assumption "delay of $y+$ is less than sum of delays of $z+$ and $x\Leftrightarrow$" is satisfied, then the optimized circuit is a correct implementation for the original specification.

We can now conclude about two potential sources of gain in optimization based on timing assumptions:

1) Unreachability of some states due to timing (*timed unreachable states*).

2) Simultaneity of transitions which gives freedom in choosing enabling regions for signals (*lazy behavior*).

In both cases the don't care space for the functions of circuit signals increases which finally leads to simpler implementations.

The idea to use don't cares coming from the timed unreachable states is due to [18, 17] and was successfully exploited in the ATACS tool for the design of timed circuits. To our knowledge the observation about the additional don't cares coming from lazy behavior appears for the first time and is the main theoretical contribution of the paper. This concept is developed in more detail in the next section.

[2]The latter can be formalized in terms of delay range for gates. If a delay range is $[gate_{min}, gate_{max}]$ then the assumption can be posed as $k * gate_{min} > gate_{max}$.

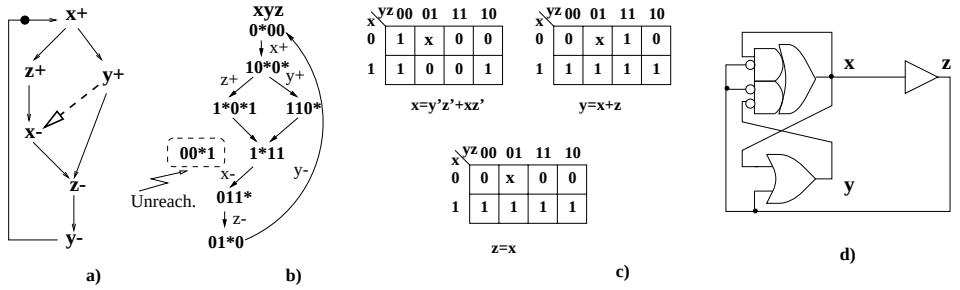

Figure 2: xyz example. Optimization by timed unreachable states.

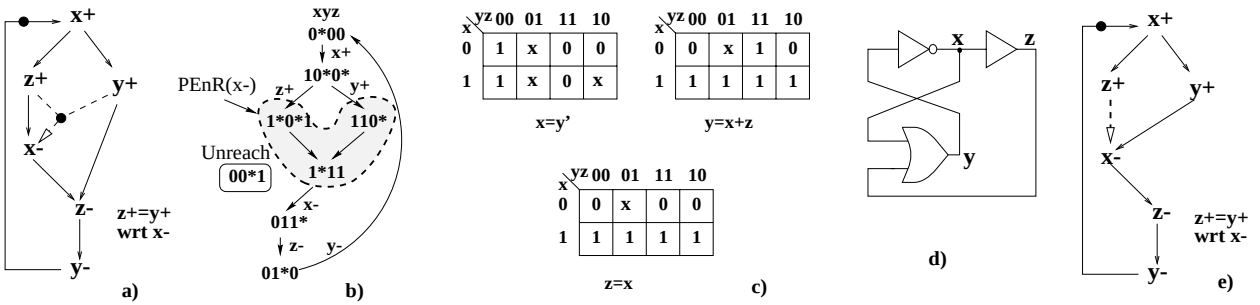

Figure 3: xyz example. Optimization by lazy behavior.

4 Lazy systems

This section introduces the basics for defining lazy systems which were informally introduced in Section 3. The main distinctive feature of a lazy system is that it considers a non-zero delay between enabling of transition and its firing. Due to this, the set of states in which a transition is enabled might be larger than the set of states in which the transition fires; recall that for speed-independent systems (cf. Section 2) these two sets always coincide since every transition can have an arbitrary delay.

4.1 Lazy State Graphs

Definition 4.1 (Enabling and firing regions) *A* potentially enabling region, $PEnR(a*)$, *of a signal transition $a*$ is a maximal connected set of states such that in all states of $PEnR(a*)$ signal a has the same current value and might be enabled.*

An enabling region $EnR(a*)$, *of a signal transition $a*$ is a maximal connected subset of states of $PEnR(a*)$ in which transition $a*$ is enabled.*

A firing region, $FR(a*)$, *of a signal transition $a*$ is a maximal connected set of states such that $a*$ can fire from any state of $FR(a*)$, i.e. $s \in FR(a*) \Rightarrow s \stackrel{a*}{\rightarrow}$.*

The difference between the notions of firing and enabling regions comes from the observation of a non-zero delay in firing a lazy transition. The need to introduce a potentially enabling region together with enabling region simply was illustrated by the optimization loop on the example of timed implementation of *xyz* STG. A potentially enabling region gives an upper bound for a set of states in which a transition can be enabled. The freedom in choosing the enabling region within the $PEnR$ gives additional possibilities for logic optimization. Note that at the specification level it is sufficient to consider firing and potentially enabling regions.

It is easy to see the following correspondence between the introduced regions: $FR(a*) \subset EnR(a*) \subset PEnR(a*)$.

Examples of potentially enabling and firing regions are illustrated by Figure 3,c, where $FR(x\Leftrightarrow) = \{111\}$ and $PEnR(x\Leftrightarrow) = \{101, 110, 111\}$. The implementation of a signal x as an inverter ($x = \overline{y}$) uses only a subset of $PEnR(x\Leftrightarrow)$ for enabling of $x\Leftrightarrow$: $EnR(x\Leftrightarrow) = \{110, 111\}$.

Definition 4.2 (Lazy transition, lazy state graph) *A transition $a*$ is called lazy if $PEnR(a*) \neq FR(a*)$.*

A state graph is called lazy (LzSG) iff:

1. for every transition $a*$ of SG the potentially enabling ($PEnR(a*)$) and firing ($FR(a*)$) regions are defined, and

2. at least one transition is lazy.[3]

The correctness properties of SGs can be easily transferred onto lazy state graphs. An LzSG will be called consistent, deterministic and commutative if the underlying SG has these properties. For persistency property the distinction between the firing and enabling regions requires to generalize its definition for LzSGs. Persistency captures the absence of hazards in an *implementation* derived by LzSG, therefore we will formulate it in terms of enabling regions rather than by $PEnRs$.

Definition 4.3 (Persistency) *A signal transition $a*$ is persistent in LzSG if two conditions are satisfied:*

- *in $EnR(a*) \subset PEnR(a*)$ no disabling of $a*$ is possible, i.e. $\forall s \in EnR(a*)$, $s \stackrel{b*}{\rightarrow} s_1$, $a \neq b$, $a*$ is enabled in s_1.*
- *no transitions from firing to the corresponding enabling region is possible, i.e. for any $b*$ there is no transition $s_1 \stackrel{b*}{\rightarrow} s_2$ such that $s_1 \in FR(a*)$ and $s_2 \in EnR(a*) \Leftrightarrow FR(a*)$.*

The following property reveals the distinctive features of firing and enabling regions of persistent transitions.

Property 4.1 *For a persistent transition $a*$ in LzSG every $EnR(a*)$ and $FR(a*)$ can be exited only by the firing of $a*$.*

[3] As we are targeted at optimization of signals that are synthesized by a circuit we will not consider lazy behaviors of input signals.

The proof is trivial. For $FR(a*)$ it follows directly from Condition 2 of Definition 4.3, while for $EnR(a*)$ exiting it by any signal different from a means the disabling of $a*$, which contradicts the persistency requirement.

Property 4.1 bridges up the conditions for hazard-free implementation of an LzSG with the similar ones for implementing an SG. It can be shown that, if the timing assumptions for an initial specification are satisfied, then any LzSG in which transitions of output signals are persistent has a hazard-free implementation with complex gates. The implementation issues for an LzSG will be discussed in detail in Section 5. Before that we should formalize timing assumptions and determine what kind of assumptions are really needed.

4.2 Timing assumptions

Timing assumptions could be defined in the form telling that one event is happening before or after another. However, this form is ambiguous for cyclic specifications because their transitions can be instantiated many times and different instances may have different ordering. More rigor in defining ordering relations can be achieved at the unfolding level [13], i.e. when an original net is unfolded into an equivalent acyclic description. The theory of timed unfoldings is however restricted to simple structural classes of STGs and the timing analysis algorithms are computationally expensive [3, 8]. We will therefore rely on a more conservative approximation of timing assumptions in LzSGs.

Difference constraints. A difference constraint $b* < a*$, involving two potentially concurrent events $a*$ and $b*$, assumes that, due to certain timing characteristics, $b*$ fires earlier than $a*$. Formally, it can be defined through the *maximum separation* $Sep_{max}(b*, a*)$ between events $b*$ and $a*$ [12]. The maximum separation gives an upper bound on the time difference between firings of $b*$ and $a*$. If $Sep_{max}(b*, a*) < 0$ then $b*$ always fires earlier than $a*$. In SG this assumption can be represented by the *concurrency reduction* of $a*$ with respect to $b*$.

Concurrency reduction can be performed in two steps:

1: Remove all arcs $s \xrightarrow{a}$ such that s is backward reachable from $PEnR(a*) \cap PEnR(b*)$ [4] (this delays $a*$ until $b*$ fires)

2. Remove unreachable states (due to delaying $a*$ against $b*$)

Let us illustrate the application of a difference constraint $b+ < d+$ on the example of STG in Figure 4,a. Concurrency reduction of $d+$ with respect of $b+$ in the SG of Figure 4,b first removes the arc $1010 \xrightarrow{d+}$, and then deletes states 1011 and 1001 which become unreachable after the arc $1010 \xrightarrow{d+}$ is removed. As the result we obtain an LzSG in Figure 4,c with a lazy signal d ($FR(d+) \neq EnR(d+)$).

Timing assumptions based on difference constraints are the main source for the elimination of timed unreachable states [18, 17] but they cannot fully express the lazy behavior of signals.

Simultaneity constraints. Exploiting simultaneity in transition firings is a key factor in the burst-mode design methodology [19]. Here, the environment is considered to be slow and therefore the skew of delays for output signals is negligible, i.e. output transitions are *simultaneous* from the point of view of environment (fundamental mode assumption). The weak point of the fundamental mode is that it must be applied to a circuit as a whole, which essentially relies on even distribution of propagation delays within the circuit. To lift this restriction we consider simultaneity assumptions more locally, and hence introduce a *local fundamental mode* with respect to particular groups of transitions. The simultaneity assumption is a *relative notion*, which is defined on a set of transitions T with respect to a reference transition $a*$. From the point of view of $a*$ the skew of firings times of transitions from T is negligible. Formally this can be defined by the following separation inequalities: $\forall b*, c* \in T,\ Sep_{max}(b*, c*) < d(a*)$, where $d(a*)$ is a lower bound delay for transition $a*$.

Let us consider the simultaneity assumption between transitions $b+$ and $c+$ with respect to $a\Leftrightarrow$ in the LzSG from Figure 4,c. This assumption influences the LzSG in two ways:

1) State 0100 which is entered when $a\Leftrightarrow$ fires before $c+$ becomes unreachable. (Indeed from $Sep_{max}(c+, b+) < d(a\Leftrightarrow)$ (coming from simultaneity assumption) and $Sep_{max}(b+, a\Leftrightarrow) < 0$ (coming from causality between $b+$ and $a\Leftrightarrow$) follows the difference constraint $Sep_{max}(c+, a\Leftrightarrow) < 0$)

2) A potentially enabling region for $a\Leftrightarrow$ is expanded in state 1010 (see Figure 4,d).

The above imples that optimization based on simultaneity assumptions goes beyond the possibilities given by difference constraints only.

Early enabling. The simultaneity assumptions exploit "laziness" between concurrent transitions. This idea can be generalized for ordered transitions as well. Suppose that transition $a*$ triggers the firing of $b*$ and we assume that in the implementation $a*$ is "faster" than $b*$ (or more formally: $D(a*) < d(b*)$, where $D(a*)$ and $d(b*)$ are the maximal and minimal delays of transitions $a*$ and $b*$ respectively). Then the enabling of $b*$ can be started earlier (from the events triggering $a*$ e.g.) and the proper ordering of $a*$ before $b*$ will be ensured by the timing properties of implementation. In LzSG this results in the expansion of $PEnR(b*)$ into the enabling region for $a*$.

An early enabling of $d\Leftrightarrow$ is illustrated in Figure 4,d.

Finally, all of the introduced timing assumptions are shown in the STG of Figure 4,e, where dashed arc $(b+, d+)$ corresponds to the difference constraint $b+ < d+$, hyperarc $(b+ c+, a\Leftrightarrow)$ corresponds to the simultaneity of $b+, c+$ with respect to $a\Leftrightarrow$, and triggering of $d\Leftrightarrow$ by $a\Leftrightarrow$ and $c\Leftrightarrow$ (instead of $b\Leftrightarrow$) shows the early enabling of $d\Leftrightarrow$ (timing arc $(b\Leftrightarrow, d\Leftrightarrow)$ is needed to keep an information about the original ordering between $b\Leftrightarrow$ and $d\Leftrightarrow$).

5 Implementation

The method presented in the previous sections has been implemented in the tool `petrify`, that can synthesize asynchronous circuits from STG specifications.

The timing assumptions on the behavior of the circuit and the environment are specified by the designer. Two types of assumptions are accepted:

- $t(a) < t(b)$, indicating that event a will always occur before event b, even they are potentially concurrent according to the original STG.
- $t(a) = t(b)$ wrt c indicating that the occurrence of a and b can be considered simultaneous with regard to event c

In the example of Figure 3, the following assumptions have been specified for optimization:

$$t(y+) < t(x\Leftrightarrow) \text{ and } t(y+) = t(z+) \text{ wrt } x\Leftrightarrow$$

The following procedure is executed to do logic synthesis of each output signal x:

1. The set of unreachable states, PEnR($x+$) and PEnR($x\Leftrightarrow$) are calculated according to the timing assumptions. DC is defined as the set of binary vectors not corresponding to any reachable state[5].
2. Those states that may introduce CSC conflicts by a possible change of the next-state function are removed from their corresponding PEnR.

[4] $PEnR(a*) \cap PEnR(b*)$ in SG gives a set of states where $a*$ and $b*$ might be concurrently enabled.

[5] DC must be only calculated once for all signals.

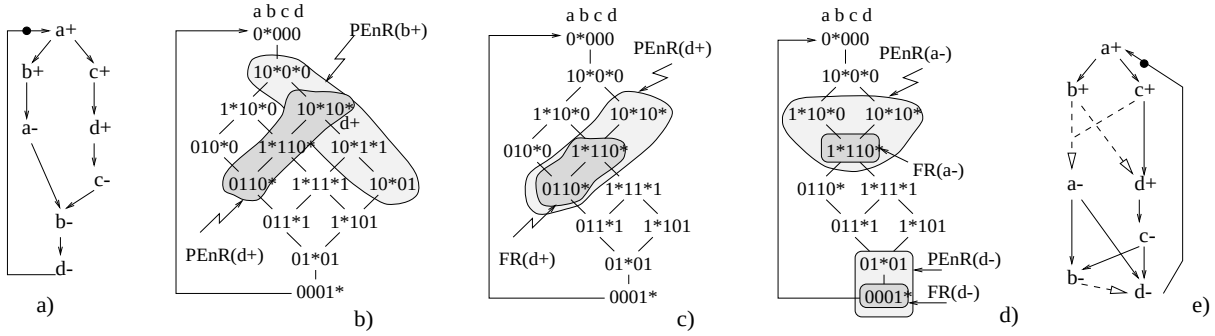

Figure 4: Application of difference c), simultaneity and early enabling d) timing assumptions

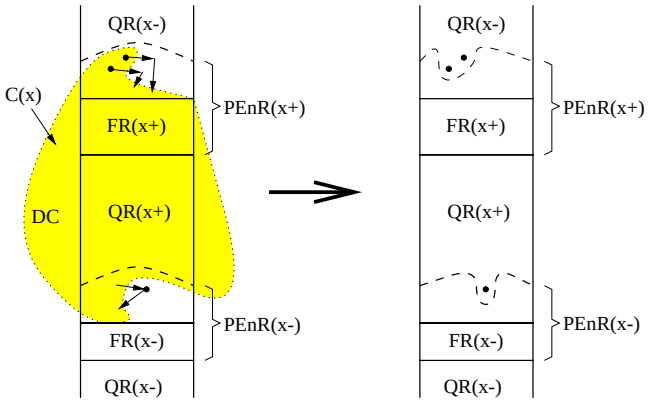

Figure 5: Reduction of PEnRs for non-monotonic covers

3. Define $\mathsf{ON}(x) = \mathsf{FR}(x+) \cup (\mathsf{QR}(x+) \setminus \mathsf{PEnR}(x\Leftrightarrow))$ and $\mathsf{DC}(x) = \mathsf{DC} \cup (\mathsf{PEnR}(x+) \setminus \mathsf{FR}(x+)) \cup (\mathsf{PEnR}(x\Leftrightarrow) \setminus \mathsf{FR}(x\Leftrightarrow))$.

4. $C(x) = \text{boolean\_minimization}(\mathsf{ON}(x), \mathsf{DC}(x))$

5. If $C(x)$ is a hazard-free cover of $\mathsf{ON}(x)$, return $C(x)$ as the cover for signal x. $\mathsf{ON}(x) \subseteq C(x)$ is a hazard-free cover if:

   $C(x) \cap \mathsf{PEnR}(x+)$ is a monotonic cover of $\mathsf{FR}(x+)$

   and

   $\overline{C(x)} \cap \mathsf{PEnR}(x\Leftrightarrow)$ is a monotonic cover of $\mathsf{FR}(x\Leftrightarrow)$

6. Otherwise, remove from $\mathsf{PEnR}(x+)$ and $\mathsf{PEnR}(x\Leftrightarrow)$ those states that violate the previous monotony conditions for $C(x)$ and $\overline{C(x)}$. This transformation is illustrated in Figure 5. Thus, the new PEnRs will be equal to or smaller than the previous ones.

7. Go to step 3

In the worst case, the loop 3-7 will converge towards a configuration with $\mathsf{PEnR}(x+) = \mathsf{FR}(x+)$, $\mathsf{PEnR}(x\Leftrightarrow) = \mathsf{FR}(x\Leftrightarrow)$, $\mathsf{ON}(x) = \mathsf{FR}(x+) \cup \mathsf{QR}(x+)$ and $C(x) = \mathsf{ON}(x) \cup d$ for some $d \subseteq \mathsf{DC}$. Note that the largest timing optimization is achieved when $C(x)$ completely covers $\mathsf{PEnR}(x+)$ and does not intersect $\mathsf{PEnR}(x\Leftrightarrow)$.

In practice, most covers $C(x)$ are monotonic after the first boolean minimization and no iteration is required. Only in some rare cases, more than two iterations are executed.

`Petrify` includes a boolean minimizer that delivers several covers with similar cost. Among them, a cost function selects those that are monotonic and have the smallest literal count.

In the future we foresee to provide more freedom to the designer to seek the best trade-off between area and performance. This can be implemented by enabling the designer to tune some parameters of the cost function.

6 Experimental results

In this section we report on the experimental setup, including a discussion on how to derive timing assumptions from knowledge about the environment and information about the circuit implementation, and we show preliminary experimental results.

6.1 Design flow and assumption derivation

The timing-based optimizations described in this paper best fit into a design flow that satisfies three requirements, in order of importance:

1. some information is known about the delay of the environment in which the circuit will operate (or, alternatively, large portions of the overall asynchronous control are synthesized and analyzed for timing properties simultaneously),

2. good control is possible over the delay of gates and wires within the circuit portion on which timing-based optimization is performed.

3. a good asynchronous timing analysis tool is available.

The first requirement is necessary in order to apply optimizations in the style of Myers [17], as extended in this paper to use don't cares instead of pre-specified values.

The second requirement is necessary in order to take maximum advantage from the capabilities of lazy timing optimization. Consider, for example, the decision to enable a slow signal early, in order to speed it up. In that case, changing the logic due to the addition of laziness to the SG may have the unwanted effect of firing this signal *too early*. Without transistor sizing or delay padding, there is little hope of closing the optimization loop in a clean and easy way, since it is very difficult to determine a priori which optimizations are *safe* and *preserve the timing assumptions on which they are based*. On the other hand, with transistor sizing or delay padding one can restore the correct ordering of transitions and ensure the validity of almost any early enabling due to separation assumptions between outputs.

The third requirement is, unfortunately, still far from realizable. Although good progress in this direction has been made [3, 12, 20, 1], we are still far from having an automated tool that can handle realistic circuits in a reasonable time in the

presence of input non-determinism. Hence for now this step is left to the designer's intuition and ability.

For this paper, we have assumed that

1. All inputs to the circuit are slower than any single gate inside the circuit. This is generally a realistic assumption even if the "apparent" behavior of those inputs is just that of a buffer or inverter, since this generally "hides" the control of some other asynchronous pipeline stage, that behaves like a simple handshake, but has actually large delays in comparison with those of the gates composing the circuit that is being designed.

2. No control over gate delays is possible. We actually used a fairly small standard cell library, in order to test our approach in a sort of worst case.

3. Performance analysis, as well as part of timing analysis, is done by *logic simulation*. We synthesized both the circuit and the environment, and artificially slow down the environment implementation by delay padding. Moreover, we limited ourselves to circuits without input non-determinism (since non-determinism is not synthesizable with standard speed-independent techniques), or chose one specific operational cycle of circuits with input non-determinism.

The results of simulation were used both to derive internal timing assumptions, for the purpose of early enabling, and in order to check that those assumptions were satisfied after lazy resynthesis. We manually verified that the result of simulation was consistent with the STG specification. This is by no means a suggested design flow choice, but it is just a temporary solution.

6.2 Experimental results

Table 1 shows the results of the application of our timing-based optimization procedure to a well-known set of asynchronous benchmark circuits. The experiment was organized as follows.

- We implemented all the circuits by using basic gates from a small library (1 not, 4 and/nand/or/nor, 4 and-or-invert, 2 S/R flip-flop and 1 C-element) based on ES2 1 μm technology,

- We ran a logic simulation of the circuit twice, once with 1 ns delay on every input signal, and once with 2 ns on every input signal. We identified the duration of a cycle in the simulation, and used it as a measure of circuit performance (in fact the simulation always converged to the critical cycle in two iterations). We used the difference between the two runs in order to isolate the contribution to the critical cycle due to the circuit from that due to the environment. The result of this first *speed-independent* synthesis run is presented in columns 2 and 3, by showing area (factored form literals) and critical cycle contribution *due to the circuit* (in picoseconds; the delay of an inverter is about 200 ps in this technology).

- We added separation assumptions stating that input signals are slower than any output or internal (state) signal, and implemented all the circuits again.

- We ran the simulation again, with 1 ns delay on all inputs, checking that the timing assumptions were satisfied. The result of this second *timed* synthesis and simulation run (again, factoring out the contribution to the period due to the inputs) is presented in columns 4 and 5, both in absolute terms and as a percentage.

- We added further separation assumptions *between outputs*, based on relative delays of gates in the implementation. The simulation done in the previous step was used in order to derive firing times of internal and external signals, and manual analysis was used in order to determine the exact timed causal relations. We implemented the circuits again. In some case, no improvement could be obtained while still satisfying the assumptions. Otherwise, the improvement with respect to timed synthesis was due both to a larger don't care space and to early enabling.

- We ran the simulation, checking the satisfaction of the assumptions. The result of this third *lazy* synthesis step is presented in columns 6 and 7.

From these *preliminary* experiments we can conclude that lazy optimization is a very promising technique for aggressive timing optimization of asynchronous control circuits, because

- it allows one to effectively achieve the same objective of increasing throughput as *pipelining* in synchronous circuits, but

- avoids (or limits) the penalty due to pipeline latches in terms of both area and performance (latency and ultimate throughput limitation due to latch internal delays).

Moreover, the technique is applicable even without sophisticated transistor sizing techniques, that would make it even more effective, and without automated timing analysis tools, that would make it easier and safer[6].

7 Conclusions

We have proposed Lazy Transition Systems, a theoretical model for timed circuit synthesis, where the notions of enabling and firing are distinguished for a signal switching event. In this new framework, we have also presented necessary conditions for synthesis of circuits with correct behavior under given timing assumptions.

We can now summarize the main results of this paper by putting our method into the overall taxonomy of issues involved in timed circuit synthesis:

- Both types of relative timing assumptions, difference (one-sided) and simultaneity (two-sided) constraints, are used.

- The objects on which timing information can be defined are either individual transition delays (they are good for locally related events; timing analysis is simple) or firing times (more global; relate sequences of events).

- The way timing determines the don't care space is either due to unreachability (they are aimed at area; higher speed is achieved as logic is simpler) or due to laziness, i.e. enabling region expansion (these are targeted for both area and performance).

- The method currently solves a "direct" problem: given an STG model with timing assumptions, obtain an optimized circuit (it would be possible to consider the "inverse" one: given an STG model, obtain an optimized circuit with timing constraints).

- Timing analysis is at present assumed to be the designer's responsibility (which is cheap and fast, local dependencies, approximate). In the future an automatic tool (still expensive, global dependencies, exact) can be used.

Preliminary experimental results confirm that significant area and speed improvements can be achieved by exploiting the extra don't care space due to the laziness of timed events. This approach helps bridging two critical gaps existing in synthesis of control circuits today. The first gap is between the two main approaches for automated asynchronous controller synthesis, those based on fundamental (global timing constraints) and input-output modes. It also tackles the traditionally unreconcilable gap between asynchronous and synchronous circuit synthesis [6]. Namely, the

[6] In this experiment we considered only a single delay number for each gate when verifying the timing assumptions by simulation, instead of considering the safer min-max delay intervals allowed by the above mentioned separation analysis techniques.

name	speed-indep.		timed (slow env.)				lazy (intern. del.)			
	area	perf.	area	%	perf.	%	area	%	perf.	%
half	11	3042	7	64	2401	79	7	64	1845	61
vbe5c.2	14	4956	14	100	4956	100	13	93	4221	85
vbe5b.2	16	4113	15	94	3979	97	13	81	2885	70
chu133	16	6001	14	88	5147	86	13	81	3604	60
hazard.2	17	7612	17	100	7612	100	16	94	6359	84
converta	18	8671	14	61	6397	74	14	61	6397	74
rcv-setup	19	7257	18	95	7069	97	18	95	7026	97
ebergen.2	22	9647	21	95	9091	94	21	95	8971	93
nak-pa	27	7642	27	100	7642	100	24	89	7222	94
nowick	29	10123	29	100	10123	100	27	93	8698	86
mp-forward-pkt	33	9839	33	100	9839	100	30	91	8795	89
ram-read-sbuf	36	10507	35	97	10437	99	30	83	8935	85
seq4.2	37	9195	35	95	7725	84	34	92	7530	82
wrdatab	39	8403	37	94	7829	93	35	89	11688	94
sbuf-ram-write	39	13805	39	100	13805	100	34	87	10323	75
mmu	40	7141	35	88	6902	97	33	82	6509	91
mr1	45	7747	44	98	6974	90	41	91	6536	84
master-read	47	6413	42	89	5250	82	40	85	4997	78
mr0	59	9869	55	93	9080	92	52	88	7249	73
pe-send-ifc	71	20119	62	87	18634	93	57	80	19150	95
total	619	172102	593	96	160822	93	552	89	148940	87

Table 1: Experimental results of lazy optimization

proposed lazy optimization technique is in many ways complementary to the techniques used for synchronous circuits for the same objective (higher throughput). Our approach thus identifies ways in which synthesis of asynchronous circuits can achieve greater practicality and wider acceptance due to its more active dealing with time information. To this end, we feel urgent need for more research in the area of mechanizing the feedback between timing optimization and timing analysis.

References

[1] T. Amon, H. Hulgaard, G. Borriello, and S. Burns. Timing analysis of concurrent systems. Technical Report UW-CS-TR-92-11-01, University of Washington, 1992.

[2] S. Burns. General conditions for the decomposition of state holding elements. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems, Aizu, Japan*, March 1996.

[3] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, and A. Yakovlev. Decomposition and technology mapping of speed-independent circuits using boolean relations. In *IEEE/ACM Int. Conference on Computer Aided Design*, pages 220–227, San Jose, USA, November 1997.

[5] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Trans. Inf. and Syst.*, E80-D(3):315–325, March 1997.

[6] M. Horowitz. Clocking for high performance processors. (Invited talk at Async'98), March-April 1998.

[7] D. A. Huffman. The synthesis of sequential switching circuits. *J. Franklin Institute*, 257:161–190,275–303, March 1954.

[8] Henrik Hulgaard and Steven M. Burns. Bounded delay timing analysis of a class of CSP programs with choice. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11, November 1994.

[9] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. John Wiley and Sons, London, 1993.

[10] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits*. Kluwer Academic Publishers, 1993.

[11] A. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communications*, The UT Year of Programming Series. Addison-Wesley, 1990.

[12] K. McMillan and D. Dill. Algorithms for interface timing verification. In *Proceedings of the International Conference on Computer Design*, October 1992.

[13] Kenneth McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In G. v. Bochman and D. K. Probst, editors, *Proc. International Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177, 1992.

[14] D. E. Muller and W. C. Bartky. A theory of asynchronous circuits. In *Annals of Computing Laboratory of Harvard University*, pages 204–243, 1959.

[15] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, April 1989.

[16] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng. Automatic synthesis and verification of gate-level timed circuits. Technical Report CSL-TR-94-652, Stanford University, January 1995.

[17] Chris J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Dept. of Elec. Eng., Stanford University, October 1995.

[18] Chris J. Myers and Teresa H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[19] Steven M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, Department of Computer Science, 1993.

[20] T. G. Rokicki and C. J. Myers. Automatic verification of timed circuits. In *Proc. International Workshop on Computer Aided Verification*, pages 468–480. Springer-Verlag, 1994.

[21] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, U.C. Berkeley, May 1992.

[22] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley Interscience, 1969.

[23] K. van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

[24] Peter Vanbekbergen, Gert Goossens, and Bill Lin. Modeling and synthesis of timed asynchronous circuits. In *Proceedings of the European Design Automation Conference (EURO-DAC)*, pages 460–465. IEEE Computer Society Press, September 1994.

[25] Chantal Ykman-Couvreur, Bill Lin, and Hugo de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, September 1994. User and Tutorial manual.

[26] Kenneth Yi Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems*. PhD thesis, Stanford University, August 1994.