Synthesis of asynchronous control circuits with automatically generated relative timing assumptions

[1]Jordi Cortadella*, [2]Michael Kishinevsky, [2]Steven M. Burns and [2]Ken Stevens
[1]Univ. Politècnica de Catalunya, Barcelona, Spain and [2]Strategic CAD Lab, Intel Corporation, USA

Abstract

This paper describes a method of synthesis of asynchronous circuits with relative timing. Asynchronous communication between gates and modules typically utilizes handshakes to ensure functionality. Relative timing assumptions in the form "event a occurs before event b" can be used to remove redundant handshakes and associated logic. This paper presents a method for *automatic* generation of relative timing assumptions from the untimed specification. These assumptions can be used for area and delay optimization of the circuit. A set of relative timing constraints sufficient for the correct operation of the circuit is back-annotated to the designer. Experimental results for control circuits of a prototype iA32 instruction length decoding and steering unit called RAPPID ("Revolving Asynchronous Pentium®Processor Instruction Decoder") shows significant improvements in area and delay over speed-independent circuits.

1 Introduction

Asynchronous communication utilizes handshaking to ensure functionality that require some area and delay penalty with respect to synchronous design. Timing information can be used to combat the full handshake overhead in area and delay by removing redundant handshakes and associated logic. Since absolute timing information is mostly unknown until layout is complete, *relative timing* information in the form "event a occurs before event b" is a natural representation of timing that can be used in the design flow.

Relative timing (RT) was used for design of a prototype iA32 instruction length decoding and steering unit called RAPPID ("Revolving Asynchronous Pentium®Processor Instruction Decoder") that was fabricated and tested successfully [15, 16]. Silicon results show significant advantages - in particular, performance of 2.5-4.5 instructions per nS - with manageable risks using this design technology. RAPPID achieves three times faster performance and half the latency dissipating only half the power and requiring a minor area penalty as a comparable 400MHz clocked circuit. Another experiment with a circuit based on timing assumptions is described in [2].

The design flow for synthesizing relative timing circuits is as follows. Relative timing assumptions are provided by the user or extracted by the algorithm presented in this paper. The circuits are then designed using the assumptions for area and delay optimization. RT circuits can be optimized with respect to the untimed circuits for two reasons:

- RT assumptions reduce the set of reachable states and hence increase the number of don't care states for logic optimization of all signals.
- It is possible to extend the set of states in which a signal is enabled without changing the set of reachable states if other enabled signals are known to be or can be made faster than the early enabled (a.k.a. lazy) signal. This additional flexibility adds local don't cares that can differ from one signal to another.

A (possibly relaxed) subset of timing assumptions used for optimization is back-annotated by the tool and become timing *constraints*. Different valid netlists require different timing constraints. The circuits are then designed to meet the relative orderings, or verified that the restrictions are already part of the delays in the system. Methods based on separation analysis [6], geometric timing [10], and relative timing can be deployed for verification [12].

In [3] it is shown that relative timing synthesis can be automated using lazy transition systems in which enabling and firing regions for signal transitions are distinguished. This paper enhances the method of [3] in three major ways.

- A method for automatic generation of timing assumptions starting from a speed-independent (untimed) specification is presented. Most of the timing assumptions used in RAPPID circuits can be automatically extracted. Only architectural or environmental assumptions on the inputs needed to be specified by the user.
- A method for automatic backannotation of RT constraints sufficient for the correct operation of a circuit is developed.
- A method for timing aware state encoding is deployed. It reduces the number of state signal and generates timing assumptions for state signals if necessary. It has a significant positive effect on both area and performance.

Section 2 presents basic theory and models. Section 3 described a method for automatic generation of RT assumptions. Section 4 presents technique for extracting timing constraints for a derived RT netlist and briefly describe timing-aware state encoding. Section 5 presents experimental results.

2 Basic notions

For brevity, we assume the reader to be familiar with Petri nets, a formalism used to specify concurrent systems. We refer to [9] for a general tutorial on Petri nets. Lazy transition systems and lazy state graphs were introduced in more detail in [3].

2.1 Transition Systems and State Graphs

A *transition system* (TS) is a quadruple [13] $TS = (S, E, T, s_{in})$, where S is a *non-empty* set of *states*, E is a set of *events*, $T \subset S \times E \times S$ is a *transition* relation, and s_{in} is an *initial state*. The elements of T are called the *transitions* of TS and will be often denoted by $s \xrightarrow{e} s'$ instead of (s, e, s').

State Graphs are binary interpreted transition systems: every state is assigned a binary vector of signal values in the specified circuit; every event is interpreted as a rising ($a+$) or falling ($a-$) transition of a signal a. Notation $a*$ is used if one is 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.

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 rising

*This work was suported by a grant from Intel Corporation and was done during a visit to SCL in summer 1998.

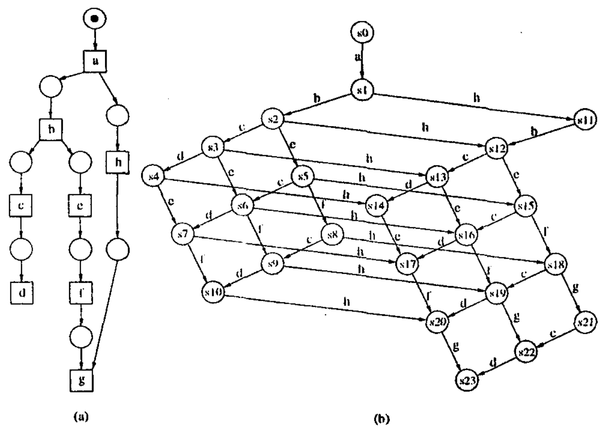


Fig. 1: (a) Petri net, (b) Transition System.

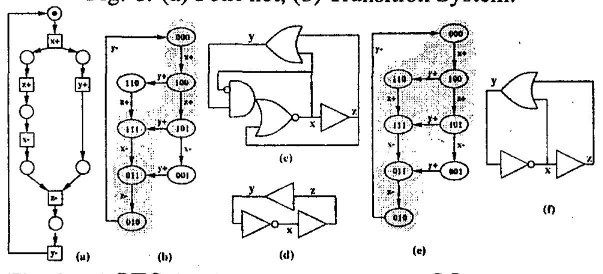


Fig. 2: (a) STG for the *xyz* example, (b,e) SGs with timing domains, (c,d,e) Circuits.

and falling transitions alternate for every signal on any path in the SG. An example of a TS and a SG are given in Figure 1.(b) and Figure 2.(b), correspondingly.

2.2 Signal Transition Graph

A Signal Transition Graph (STG) is a Petri net (PN) in which transitions are labeled with rising and falling signal transitions like in a SG. An example of a PN is shown in Figure 1.(a). This PN corresponds to a TS in Figure 1.(b). 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. Figure 2.(a) depicts an STG with three signals, x,y, and z corresponding to the SG in Figure 2.(b). For simplicity, places with only one input and output transitions are often omitted in STGs.

2.3 Excitation and quiescent regions

The *excitation region* of an event a^*, denoted by $\mathrm{ER}(a^*)$, is the set of states such that $s \in \mathrm{ER}(a^*) \Leftrightarrow s \xrightarrow{a^*}$. The *quiescent region* of $a+$, denoted by $\mathrm{QR}(a+)$, is the set of states such that $s \in \mathrm{QR}(a+) \Leftrightarrow v_a(s) = 1 \wedge s \notin \mathrm{ER}(a-)$. Similarly, $s \in \mathrm{QR}(a-) \Leftrightarrow v_a(s) = 0 \wedge s \notin \mathrm{ER}(a+)$. In Figure 2.(b), $\mathrm{ER}(x-) = \{101, 111\}$ and $\mathrm{QR}(x-) = \{001, 011, 010\}$.

2.4 Lazy transition systems

The main distinctive feature of a lazy system is that it can assume 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.

Definition 2.1 (Enabling and firing regions) *The* enabling region, $\mathrm{EnR}(a*)$, *of a signal transition $a*$ is a the set of states in which transition $a*$ is enabled. The* firing region, $\mathrm{FR}(a*)$, *of a signal transition $a*$ is the set of states from which $a*$ can fire, i.e. $s \in FR(a*) \Leftrightarrow \exists s' : s \xrightarrow{a*} s'$.*

A *potentially enabling region*, PEnR, gives an upper bound for a set of states which can be selected as an actual enabling region in the RT-implementation. The freedom in choosing the enabling region within the PEnR gives additional possibilities for logic optimization. It is easy to see the following correspondence between the introduced regions: $\mathrm{FR}(a*) \subseteq \mathrm{EnR}(a*) \subseteq \mathrm{PEnR}(a*)$. We will defer discussion of examples until Sections 4.1-4.2.

Definition 2.2 (Lazy state graph) *A transition $a*$ is called lazy if* $\mathrm{EnR}(a*) \neq \mathrm{FR}(a*)$. *A state graph is called lazy (*lazy SG*) iff at least one transition is lazy* [1].

The correctness properties of SGs can be easily transferred onto lazy SGs. A lazy SG is consistent, deterministic and commutative if the underlying SG has these properties. Persistency property must be generalized for enabling and firings as discussed in Section 2.8.

2.5 Timing assumptions

Timing assumptions could be conservatively defined in the form telling that one event is happening before or after another.

Difference assumptions. A difference assumption $b* < a*$ (reads $b*$ before $a*$), involving two potentially concurrent events $a*$ and $b*$, assumes that, due to certain timing characteristics, whenever $b*$ and $a*$ are both enabled, $b*$ always fires earlier than $a*$. In an SG this assumption can be represented by the *concurrency reduction* of $a*$ with respect to $b*$. RT difference assumptions allows one to eliminate states unreachable in timing domain similar to state elimination based on absolute timing information in [10, 11]. They are not sufficient however for expressing lazy behavior of signals.

Early enabling assumption. Suppose that transition $a*$ triggers the firing of transition $b*$, i.e. $a*$ and $b*$ are ordered in the specification. Assume that $a*$ can be made "faster" than $b*$ in the circuit. Then the enabling of $b*$ can be started earlier, e.g., from the events triggering $a*$, and the proper ordering of $a*$ before $b*$ will still be ensured by the timing properties of the implementation. In lazy SG this results in the backward expansion of $\mathrm{PEnR}(b*)$ into $\mathrm{FR}(a*)$.

Simultaneity assumption. The simultaneity assumption is a *relative notion*, which is defined on a set of concurrent transitions T with respect to a reference transition $a*$. It tells that from the point of view of $a*$ the skew of firings times of transitions from T is negligible. This assumption can be viewed as a *local fundamental mode* of T with respect to a and hence as a generalization of burst-mode machines [14, 17]. An example of the application of simultaneity assumption is discussed in Section 4.2.

Assumptions relating only input events cannot be automatically generated from the circuit behavior and can be provided by the designer or generated from the implementation of the environment.

2.6 Next-state functions

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. For SGs it is defined as follows:

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

[1] As we are targeted at optimization of output signals of a circuit lazy behaviors of input signals is not considered.

The next-state function f_a is correctly defined when the SG has the CSC property, i.e. there is no pair of reachable states (s,s') such that $v(s) = v(s')$ and $(s \in \mathrm{ER}(a+) \cup \mathrm{QR}(a+)$ or $s' \in \mathrm{ER}(a-) \cup \mathrm{QR}(a-))$. Note that f_a is an incompletely specified function with a *don't care* (DC) set corresponding to those binary vectors without any associated state in the SG. The logic netlist is *speed-independent* if SG is deterministic, commutative and output-persistent[4].

In the SG of Figure 2.(b), the DC set is empty since all binary vectors have a corresponding state in the SG. As an example, $f_x(101) = 0; f_y(101) = f_z(101) = 1$ since signals x and y are enabled, and z is stable in that state. The Karnaugh maps for the next-state functions are depicted in Figure 3.(a).

For a lazy SG the next-state functions are defined differently:

$$f_a(Z) = \begin{cases} 1 & \text{if } \exists s \in \mathrm{FR}(a+) \cup \mathrm{QR}(a+) \text{ s.t. } v(s) = Z \\ 0 & \text{if } \exists s \in \mathrm{FR}(a-) \cup \mathrm{QR}(a-) \text{ s.t. } v(s) = Z \\ - & \text{otherwise} \end{cases}$$

Note that this definition generally gives more don't care vectors that the definition for a SG due to two reasons:

- More states are unreachable, since timing assumption can reduce concurrency
- States in (PEnR – FR) do not belong to either FR, or QR, and hence are included into the DC-set.

As an example, in the lazy SG of Figure 2.(e), $f_x(101) = -; f_y(101) = f_z(101) = 1$ as explained in Section 4.2.

The conditions for speed-independent implementability can be trivially extended to lazy SGs.

2.7 Logic synthesis

From the next-state functions of a SG, a speed-independent circuit can be derived by implementing the boolean equation of each output signal as an atomic complex gate [8] or as a generalized C-elements [1, 7]. For example, a speed-independent complex gate implementation for the STG in Figure 2.(a) is a netlist:

$$x = \overline{z + \bar{x}y}; \quad y = x + z; \quad z = x + z\bar{y}.$$

Similarly, from the next-state function specification corresponding to a lazy SG, an RT-circuit can be derived in the form of complex gates or generalized C-elements as illustrated by an example in Sections 4.1-4.2.

2.8 Monotonic covers

Not every logic function derived from the definition of the next-state function satisfies hazard-freedom conditions, and hence valid. The following definition is related to hazards in the behavior of asynchronous circuits.

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

$$(s \in S_1 - S_2 \Rightarrow s' \in S_1) \wedge (s \in S_2 \Rightarrow s' \notin S_1 - S_2)$$

Only monotonic covers of FRs can be selected as EnRs for hazard-free solutions for logic netlist [3]. If $S_1 = \mathrm{EnR}(a*)$ and $S_2 = \mathrm{FR}(a*)$, then (1) no disabling of $a*$ is possible and (2) there are no transitions from $\mathrm{FR}(a*)$ to $\mathrm{EnR}(a*) - \mathrm{FR}(a*)$, i.e., no disabling of firings for $a*$ is possible either. Hence, persistency of $a*$ in the RT implementation is guaranteed. For example, in the SG of Figure 2.(b), the set $\{101, 110, 111\}$ is a monotonic cover of $\mathrm{ER}(x-)$. However, the set $\{100, 101, 111\}$ is not, since the transition $100 \xrightarrow{y+} 110$ violates the conditions for monotonicity.

3 Automatic generation of relative timing assumptions

3.1 Ordering relations

Let $TS = (S, T, E, s_0)$ be a transition system. Assume that every event in E corresponds to a single connected excitation region.

Definition 3.1 (Conflict) *An event* $e_1 \in E$ disables *another event* $e_2 \in E$ *if* $\exists s_1 \xrightarrow{e_1} s_2$ *such that* $s_1 \in \mathrm{ER}(e_2)$ *and* $s_2 \notin \mathrm{ER}(e_2)$. *Two events* $e_1, e_2 \in E$ *are in* direct conflict *if* e_1 *disables* e_2 *or* e_2 *disables* e_1.

Definition 3.2 (Concurrency) *Two events* $e_1, e_2 \in E$ *are concurrent (denoted by* $e_1 \parallel e_2$*) if they form a state diamond, i.e.*

1. $\mathrm{ER}(e_1) \cap \mathrm{ER}(e_2) \neq \emptyset$,

2. $\forall s \in \mathrm{ER}(e_1) \cap \mathrm{ER}(e_2) : (s \xrightarrow{e_1} s_1) \in T \wedge (s \xrightarrow{e_2} s_2) \in T \Rightarrow \exists s_3 \in S : (s_1 \xrightarrow{e_2} s_3) \in T \wedge (s_2 \xrightarrow{e_1} s_3) \in T$.

Definition 3.3 (Trigger) *An event* $e_1 \in E$ *triggers another event* $e_2 \in E$ *(denoted by* $e_1 \longrightarrow e_2$*) if* $\exists s_1 \xrightarrow{e_1} s_2$ *such that* $s_1 \notin \mathrm{ER}(e_2)$ *and* $s_2 \in \mathrm{ER}(e_2)$.

Definition 3.4 (Enabled before) *Let* $e_1, e_2 \in E$ *be two concurrent events.* e_1 can be enabled before e_2 *(denoted by* $e_1 \lhd e_2$*) if* $\exists s_1 \rightarrow s_2$ *such that* $s_1 \in \mathrm{ER}(e_1) - \mathrm{ER}(e_2)$ *and* $s_2 \in \mathrm{ER}(e_1) \cap \mathrm{ER}(e_2)$.

Definition 3.5 (Enabled simultaneously) *Let* $e_1, e_2 \in E$ *be two concurrent events.* e_1 *and* e_2 can be enabled simultaneously *(denoted by* $e_1 \Diamond e_2$*) if* $\exists s_1 \rightarrow s_2$ *such that* $s_1 \notin \mathrm{ER}(e_1) \cup \mathrm{ER}(e_2)$ *and* $s_2 \in \mathrm{ER}(e_1) \cap \mathrm{ER}(e_2)$.

Definition 3.4 can be extended to sets of events as follows.

Definition 3.6 (Enabled before a set of events) *Let* $e \in E$ *be an event pairwise concurrent with all the events in the set* $X = \{e_1, \ldots, e_n\} \subset E$. e can be enabled before X *(denoted by* $e \lhd X$*) if* $\exists s_1 \xrightarrow{e'} s_2$ *such that* $s_1 \in \mathrm{ER}(e) - \mathrm{ER}(X)$, $s_2 \in \mathrm{ER}(e) \cap \mathrm{ER}(X)$ *and* $e' \notin X$, *where* $\mathrm{ER}(X) = \mathrm{ER}(e_1) \cup \ldots \cup \mathrm{ER}(e_n)$.

Figure 1.b depicts the transition system derived from the Petri net of Figure 1.a. The following facts can be derived using the definitions above: $\neg(a \parallel b)$, $c \parallel f$, $c \lhd f$, $c \Diamond e$, etc. Event d cannot be enabled before $\{e, f\}$, but can be enabled before $\{e, f, g\}$ since there is a transition $s_9 \xrightarrow{h} s_{19}$ such that $s_9 \in \mathrm{ER}(d) - \mathrm{ER}(\{e, f, g\})$, $s_{19} \in \mathrm{ER}(d) \cap \mathrm{ER}(\{e, f, g\})$ and $h \notin \{e, f, g\}$.

3.2 Delay model

A delay model for events presented in this section gives an *informal intuitive motivation* for the automatic generation of timing assumptions. This model refers to the delay of the events in the TS. The delay of an event is defined as the difference between its enabling time and its firing time. Three types of events are considered:

Non-input events: its delay is in the interval $[1 - \varepsilon, 1 + \varepsilon]$

Fast input events: its delay is in the interval $(1 + \varepsilon, \infty)$

Slow input events: its delay is in the interval $[\Delta, \infty)$

The synthesis approach also assumes that (1) the delay of a gate implementing a non-input event can be lengthened by delay padding or transistor sizing, (2) the delay of two gates can always be made longer than the delay of one gate.

Hence, one can assume that $\varepsilon < 1/3$, (3) the circuit will never take longer than Δ time units (minimum delay of a slow input event) in becoming stable from any state of the system and a quiescent environment.

The previous assumptions on the timing behavior of the circuit can be translated into assumptions on the firing order of the events.

3.3 Rules for deriving timing assumptions

We present rules for deriving timing assumptions in the following format: (1) ordering relations that must be satisfied in a (Lazy) SG for a rule to be applied, (2) automatic timing assumption that can be generated, and (3) informal justification of a rule based on the above delay model.

3.3.1 Assumptions between non-input events

The following rules can be applied for deriving timing assumptions between non-input events, $e_1, e_2, e_3 \in E$:

I. Event enabled before another event.
Ordering relations: $(e_1 \| e_2) \wedge (e_1 \lhd e_2) \wedge (e_2 \ntriangleleft e_1) \wedge (e_1 \not\diamond e_2)$.
Difference timing assumption: e_1 fires before e_2
Delay assumptions: one gate shorter than two gates.

II. Events simultaneously enabled.
Ordering relations: $(e_1 \| e_2) \wedge (e_1 \diamond e_2) \wedge (e_2 \ntriangleleft e_1)$.
Difference timing assumption: e_1 fires before e_2
Delay assumptions: delay of e_2 longer than delay of e_1.

III. Event triggered by events simultaneously enabled.
Ordering relations: $(e_1 \| e_2) \wedge (e_1 \ntriangleleft e_2) \wedge (e_2 \ntriangleleft e_1) \wedge [(e_1 \Rightarrow e_3) \vee (e_2 \Rightarrow e_3)]$.
Simultaneity timing assumption: e_1 and e_2 simultaneous wrt e_3.
Delay assumptions: one gate shorter than two gates.

IV. Early (speculative) enabling for ordered events.
Ordering relations: $(e_1 \longrightarrow e_2)$.
Early enabling timing assumption: e_1 fires before e_2 (but e_2 can be enabled concurrently with e_1).
Delay assumptions: delay of e_1 shorter than delay of e_2.

Let us illustrate the previous cases with the example of Figure 1 assuming that all events are non-input. Timing assumptions of type I can be derived for the pairs of events (c,f), (c,g) and (e,d), where the first element of the pair is assumed to fire before the second. Timing assumptions of type II can be applied to the pairs (b,h) and (c,e). Timing assumptions of type III can be applied, e.g., to the events triggered by the pair (b,h) that triggers the events c, e and g. Timing assumptions of type IV can be applied, e.g., to the event d triggered by the event c. If this assumption applies, then potential enabling region for d includes states $\{s2, s5, s8, s12, s15, s18, s21\}$ as don't care states for the values of the next state function for signal d in addition to the originally present states $\{s3, s6, s9, s13, s16, s19, s22\}$.

3.3.2 Assumptions between non-input and input events

Assume that $e_1, e_2 \in E$ are a non-input and an input event respectively and they are concurrent.

V. Input not enabled before non-input event.
Ordering relations: $(e_1 \| e_2) \wedge e_2 \ntriangleleft e_1$.
Difference timing assumption: e_1 fires before e_2.
Delay assumptions: delay of environment longer than delay of one gate.

This assumption covers the ones of type I and II for the case in which e_2 is an input event. The delay assumption used in this case states that the response time of the environment will always be longer than the delay of one gate.

3.3.3 Assumptions between non-input events and slow input events

Assume that $e \in E$ is a *slow* input event, $X = \{e_1, \ldots, e_n\} \subset E$ is a set of non-input events and e is pairwise concurrent with all the events in X.

VI. Slow environment not enabled before non-input events.
Ordering relations: $(\forall e_i \in X : e \| e_i) \wedge e \ntriangleleft X$.
Difference timing assumptions: X fires before e.
Delay assumptions: delay of slow input event longer than Δ (delay of stabilizing the circuit under a quiescent environment).

To illustrate the meaning of this timing assumption we will consider that h is an input event and d is a slow input event in the example of Figure 1. The rest of events are non-input. After firing the events a, b and c a state in which d, e and h are enabled is reached (state s_3). At this point it can be assumed that e and f will fire before d (two gate delays vs. slow environment). However, no assumptions can be made about the firing order between d and g since g is preceded by an input event (h) for which no upper bound on its delay can be assumed. In case h would be a non-input event, d would be assumed to fire before h and g also.

4 Backannotation of timing constraints

After logic synthesis, the validity of the timing assumptions must be verified or validated to ensure the correct function of the circuit. However, the circuit may be correct for a set of states larger than the one defined by the timed domain, which can be obtained by a set of less stringent timing assumptions. In other words, some of the timing assumptions are redundant for a particular logic synthesis solution, while some other can be relaxed. This section attempts to answer the following question:

> *Can we derive a minimal set of timing assumptions sufficient for a circuit to be correct?*

This set of timing assumptions backannotated for a given logic synthesis solution is called *timing constraints*. Timing assumptions (both manual and automatic) are part of the specification and provide additional freedom for logic synthesis, while timing constraints is a part of the implementation, since they constitute requirements to be met *sufficient* for a particular netlist solution to be valid.

4.1 Example 1

Let us analyze the example in Figure 2. The shadowed states in SG of Figure 2.(b) correspond to the timed domain determined by the timing assumptions

$$z+ < y+ \qquad \text{and} \qquad y+ < x-$$

Under these assumptions, logic synthesis can be performed by considering the states 110 and 001 unreachable, i.e. in the don't care set of the logic functions for all signals x, y, z.

The circuits of Figures 2.(c) and 2.(d) have a correct behavior under the previous assumptions. Looking at the circuit of Figure 2.(c) we observe that:

- The gates $x = \overline{z + \overline{x}y}$ and $y = x + z$ are correct implementations for the whole untimed domain.
- The gate $z = x$ is a correct implementation for all the states except for 001. In this state $x = 0$ and $z-$ should have been enabled according to the next state function of the implementation, but it is not enabled in this state according to the original state graph specification.

Thus, even the circuit may have been obtained using the two previous assumptions, only one relative timing constraint $y+ < x-$ must be ensured for the circuit to be correct. In general, each gate of the circuit is correct for a subset of the untimed domain which is also a superset of the timed domain. The circuit is correct for those states in which all gates are correct.

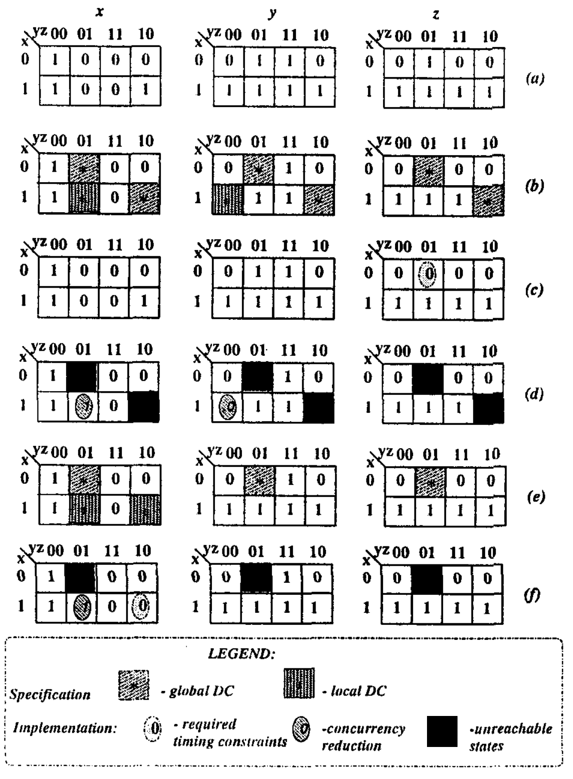


Fig. 3: Next state functions for *xyz* example: (a) Original untimed specification; (b) Specification for RT assumptions "z+ < y+ and y+ < x−"; (c,d) Implementations from Figures 2.(c,d); (e) Specification for RT assumption "y+,z+ simultaneous with respect to x−"; (f) Implementation from Figure 2.(f).

4.2 Example 2

Let us consider the same example under a simultaneity assumption "$x+$ and $y+$ are simultaneous with respect to $x-$". Under this assumption state 001 is unreachable and becomes a don't care for all signals. In addition states 101 and 110 becomes don't cares for signal x, since both belong to the potential $\mathsf{EnR}(x-)$ according to the semantics of the simultaneity assumption. Only one timing constraints, $z+ < x-$, is sufficient for the circuit in Figure 2.(f) to be correct. Gate $x = \bar{y}$ is not enabled in 101, hence concurrency is reduced in this state with respect to the original untimed SG and state 001 becomes unreachable under any gate delays. State 110 on the contrary corresponds to the concurrency expansion for enabling of $x-$. This enabling is lazy since $110 \in \mathsf{EnR}(x-) \wedge 110 \notin \mathsf{FR}(x-)$.

Figure 3 shows Karnaugh maps for the next state funstions of signals x,y, and z for specifications and implementations corresponding to the examples above. A legend shows that timing assumptions provide two types of don't care vectors in RT specifications: global don't cares corresponding to states unreachable due to timing assumptions, and local don't cares that differ for different signals. In the RT implementations some states become unreachable due to untimed concurrency reduction and therefore discrepancies in the corresponding values of the next state functions compared with the original untimed specification can be ignored; some discrepancies corresponds to concurrency reduction (disabling of signal transitions without persistency violation), and finally, other discrepancies correspond to lazy enabling and require timing constraints for correct circuit behavior.

4.3 Correctness of RT circuit

Let S be an original untimed SG with a finite set of reachable states $\mathcal{U}$ [2] and initial state s_0. Let G be a circuit netlist implementing S under timing constraints C. A pair $< G, C >$ is called a *relative timing circuit (RT circuit)*. It defines a lazy SG, $L_{<G,C>}$, with a set of reachable states $\mathcal{U}_L$. The RT-circuit implementation can contain more signals than the original specification S if some state signals are inserted for resolving state conflicts. Let us assume that S has n signals and L has $k, k \geq n$, signals. Then for comparing states one needs to use a homomorhism $h : B^k \mapsto B^n$, that given an implementation state hides $(k-n)$ new internal signals and obtains a specification state. Homomorhism, h, is naturally extended to sets of states.

A RT-circuit is said to be *correct* if the following conditions are satisfied:

1. $h(\mathcal{U}_L) \subseteq \mathcal{U}$, i.e. no states outside original untimed domain are reachable by the RT-circuit.
2. All signals persistent in S are also persistent in lazy SG $L_{<G,C>}$. All state signals inserted in $L_{<G,C>}$ are persistent. Commutativity and determinism are preserved.
3. The initial state is preserved with respect to the I/O interface, i.e., if $s_0 \in S$ and $s'_0 \in L_{<G,C>}$ are the initial states of the original SG and the lazy SG corresponding to the implementation, then there is a path $s_0 \xrightarrow{\tau} h(s'_0)$ or $h(s'_0) \xrightarrow{\tau} s_0$ in S such that sequence τ contains only events of internal signals, *not observable* by the environment.
4. No events disappear: If $\mathsf{ER}_S(e) \neq \emptyset$, then $\mathsf{FR}_L(e) \neq \emptyset \wedge h(\mathsf{FR}_L(e)) \subseteq \mathsf{ER}_S(e)$
5. No new deadlock states appear in $L_{<G,C>}$.

4.4 Theory for backannotation

For the ease of exposition let us assume that no state signals are inserted in the RT circuit, and therefore the number of signals stays the same for S and L. We will briefly discuss how state signal insertion is done in Section 4.8. Let $\mathcal{U}$ be the set of states reachable in the untimed domain of a state graph and $\mathcal{T} \subseteq \mathcal{U}$ the set of states reachable under a set of timing assumptions, manual - provided by the user and automatic - derived for synthesis according to the rules of Section 3. Let us assume that we have a circuit with m output signals, $a_1, \ldots, a_m$. Let $G = \{g_{a_1}(X), \ldots, g_{a_m}(X)\}$ (where X is the set of signals) be a set of gates implementing the RT circuit, where $g_{a_i}(X)$ denotes the boolean function implemented by the gate of signal a_i.

Reachable states in the untimed domain

Let us call $\mathcal{R}(G)$ the set of states reachable in the untimed domain for the circuit G. Note that, in general, $\mathcal{U} - \mathcal{R}(G) \neq \emptyset$ due to the reduction of concurrency imposed by the circuit, and $\mathcal{R}(G) - \mathcal{U} \neq \emptyset$ due to expansion of concurrency for enabling for lazy transitions. The latter states are not generated by our procedure since they must be unreachable in RT domain anyway. The former states are of interest, since they do not require any timing constraints (see examples 4.1 and 4.2). Let us denote $\mathcal{U}_G = \mathcal{R}(G) \cap \mathcal{U}$. $\mathcal{U}_G$ can be calculated as follows:

1. For each output signal a_i, calculate *disabled*$(a_i) = \{s \in \mathcal{U} \mid s \notin \mathsf{EnR}_G(a_i) \wedge s \in \mathsf{ER}_S(a_i)\}$, i.e. states in

[2]Our implementation is currently limited by the bounded untimed STGs and SGs. It can be easily extended to unbounded untimed STGs by making unbounded (infinitely growing) markings of STGs unreachable in RT domain.

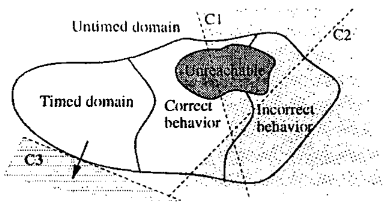


Fig. 4: Formulation of the backannotation problem. $\{C_1, C_2, C_3\}$ is the set of timing constraints sufficient for correctness of RT solution.

which a_i was enabled in the untimed domain in SG, S, but made stable by the circuit.

2. For each output signal a_i, remove all arcs $s \overset{a_i*}{\rightarrow}$ from the SG for all states $s \in disabled(a_i)$.
3. Calculate the new set $\mathcal{U}_G = \mathcal{R}(G) \cap \mathcal{U}$ of reachable states.

States with incorrect behavior

Let us call $incorrect(G) \subset \mathcal{U}_G$ the set of states *inside* $\mathcal{U}_G$ that are required to be unreachable for the correctness of the circuit. These states can be calculated as follows:

1. For each output signal a_i, calculate $incorrect(a_i) = \{s \in (\mathcal{U} - \mathcal{T}) \mid s \in \mathrm{EnR}_G(a_i) \wedge s \in \mathrm{QR}_S(a_i)\}$, i.e. states in which a_i was stable in the untimed domain, but enabled in the circuit.
2. $incorrect(G) = \mathcal{U}_G \cap \left(\bigcup_{a_i} incorrect(a_i)\right)$

Backannotation: problem formulation

We need a set of constraints that make the states in $incorrect(G)$ unreachable. A trivial solution to this problem is to take the complete set of timing assumptions used for logic synthesis, i.e. those for which $\mathcal{T}$ is the set of reachable states. Our goal, however, is to find the less stringent set of constraints sufficient to make the circuit correct. Given a set of timing constraints $C = \{C_1, \ldots, C_p\}$, we will call $\mathcal{R}(C) \subseteq \mathcal{U}$ the set of states reachable after applying C in the untimed domain. In general, the problem can be formulated as follows (see Figure 4):

> *Find a set of constraints C with the largest $\mathcal{R}(C)$ such that*
>
> *1.* $\mathcal{T} \subseteq \mathcal{R}(C) \subseteq \mathcal{U} - \text{incorrect}(G)$
>
> *2.* $\forall s \in \mathcal{T} : (s \in \mathrm{EnR}_G(a_i*) \wedge s \notin \mathrm{ER}_S(a_i*)) \Rightarrow \exists a_j : s \overset{a_j*}{\rightarrow} s' \wedge s' \in \mathcal{T} \wedge (a_j* < a_i*) \in C$

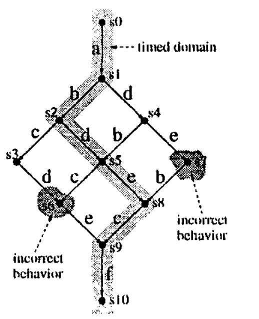
The first condition guarantees that no incorrect states *inside* $\mathcal{U}$ are reachable (constraints C_1, C_2 in Figure 4), whereas the second makes sure that no states *outside* $\mathcal{U}$ can be reached in the RT circuit (constraint C_3 in Figure 4).

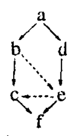
4.5 Finding a set of timing constraints

Relative timing constraints are defined in terms of firing order of events. Constraining a firing order between a pair of events makes only sense when they can be enabled simultaneously and fire in any order, i.e. when they are concurrent. Thus, each timing constraint C_i can be denoted by an ordered pair of concurrent events, e.g. $C_i = (e_j < e_k)$.

Given a constraint $C_i = (e_j < e_k)$, we define the set of arcs $disabled(C_i)$ as

$$disabled(C_i) = \{s \overset{e_k}{\rightarrow} s' \mid \exists s \rightarrow s_1 \rightarrow \ldots \rightarrow s_n : s_1, \ldots, s_{n-1} \in \mathrm{ER}(e_k) \wedge s_n \in \mathrm{ER}(e_k) \cap \mathrm{ER}(e_j)\}$$





order	unreachable
b<d	**{s4,s7}**
b<e	**{s7}**
c<d	{s4,s5,s7,s8}
c<e	{s7,s8}
d<b	{s2,s3}
d<c	**{s3}**
e<b	{s2,s3,s5,s6}
e<c	**{s3,s6}**

Fig. 5: Example for backannotation with table of unreachable states for each pair of ordered events.

In particular, the path $s_1 \rightarrow \ldots \rightarrow s_n$ can be empty if $s \in \mathrm{ER}(e_j) \cap \mathrm{ER}(e_k)$. $disabled(C_i)$ is the set of arcs with label e_k that must not fire in order for e_j to fire before e_k, i.e. those arcs with source states in which both events are concurrent or preceding $\mathrm{ER}(e_j) \cap \mathrm{ER}(e_k)$ inside $\mathrm{ER}(e_k)$.

Given a set of constraints $C = \{C_1, \ldots, C_p\}$, $\mathcal{R}(C)$ is the set of reachable states after removing the arcs in

$$\bigcup_{C_i \in C} disabled(C_i)$$

4.6 Example 3

Figure 5 shows an example for deriving a set of timing constraints for backannotation. Initially we have $\mathcal{U} = \{s_0, \ldots, s_{10}\}$ and $\mathcal{T} = \{s_0, s_1, s_2, s_5, s_8, s_9, s_{10}\}$. Let us assume that s_6 and s_7 are the states in which the behavior of the circuit is incorrect. The table in Figure 5 contains the set of states that become unreachable by reducing the concurrency between each pair of concurrent events[3]. For example, by imposing the order $d < b$, the states s_2 and s_3 become unreachable.

The problem to be solved is the following: find a set of ordering constraints between pairs of events such that the new set of reachable states covers $\mathcal{T}$ and does not intersect the set of incorrect states $\{s_6, s_7\}$. Moreover, we want to *maximize the set of reachable states*, i.e. to find the less stringent set of timing constraints.

The problem can be posed as a *covering problem*. The cells of the table in bold correspond to those constraints that do not remove any state from $\mathcal{T}$. The covering problem can be formulated as follows:

$$(e < c) \wedge (b < d \vee b < e)$$

with the minimum-cost solution $C = \{e < c, b < e\}$ and $\mathcal{R}(C) = \{s_0, s_1, s_2, s_4, s_5, s_8, s_9, s_{10}\}$

4.7 Solving the covering problem

The covering problem for backannotation does not correspond to a unate covering problem, since the cost of the final solution (number of disabled arcs) is not the sum of the cost of each constraint.

Currently, `petrify` uses a greedy approach to solve the covering problem that can be easily implemented by symbolic BDD-based techniques. It merely consists in choosing the constraint that removes the maximum number of arcs whose destination is in $incorrect(G)$ and that have not been removed by previous constraints. This process is iteratively repeated until all incorrect states become unreachable.

[3]For simplicity, unreachable states are reported in the table for this example. In general, the analysis must be performed by calculating the removed *disabled* arcs. In this particular case, the resulting analysis is the same.

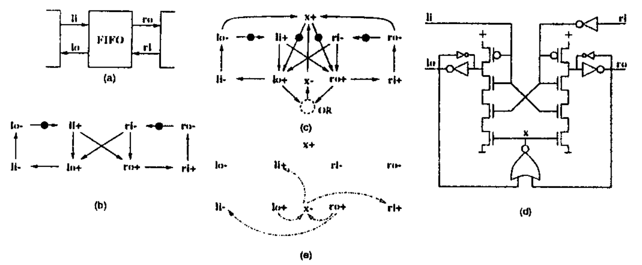


Fig. 6: (a) FIFO controller, (b) Specification, (c) Specification with state encoding signal, (d) RT implementation with gC elements, (e) Timing constraints sufficient for correctness.

In some cases, not all the incorrect states can be made unreachable since the timed state space has been produced by early enabling some events. In those cases, a similar iterative process is executed to cover those incorrect states that can be legalized by early enabling. As an example, consider the state s_6 in Figure 5. Assume that s_6 is incorrect since the next-state function indicates that f is enabled in that state. The state could be made correct by extending $\mathsf{EnR}(f)$ towards s_6 and imposing the type-IV constraint $e < f$.

4.8 Timing aware state encoding

The problem of state encoding is in inserting state signals for resolving CSC conflicts. State encoding in our implementation is automatically solved using an extension of the method presented in [4]:

- Only those encoding conflicts reachable in the RT domain are considered in the cost function such that no effort is invested in solving conflicts unreachable in RT domain, $\mathcal{T}$.
- Automatic timing assumptions can be generated for inserted state signals using rules from Section 3 implying that the state signals can be implemented as RT logic.

5 Experimental results

5.1 Academic examples

The results for the well-known benchmarks used at academia are presented in Table 1. Tables 1.(a) and 1.(b) present the results for specifications with and with *state coding conflicts* respectively. **SI**$_a$, **SI**$_t$ and **TI** represent area and delay optimization for speed-independent design, and relative timing results, correspondingly.

For each experiment, area is estimated as the number of literals of the *set* and *reset* networks of generalized C elements. Delay (response time) is estimated as the average number of non-input events in the critical path between the firing of two input events. Comparing the columns **SI**$_t$ and **TI**, we observe a reduction of about 40% in area. The reduction in response time is less than 5% if we consider all events to have a delay of one time unit. However, the performance improvement is much more pronounced if it were evaluated with actual delays, given that the logic of the timed implementation is much simpler. We report this analysis in Section 5.2.

5.2 Example: a FIFO controller

In this section we trace the development of a FIFO cell (specified in Figures 6.(a),(b)), a simplified abstraction of a part of the RAPPID design. The modules at the left and right sides of the controller have a similar speed as the controller itself. In fact, these events are generated by twin modules connected at each side. For this reason, it is not wise to assume that the input events are slow.

We simulated four FIFOs using different implementations of the FIFO cell and measured a cycle time of the FIFO and a forward latency (an average event propagation time from *li* to *ro*) of a cell. The results normalized to the delay of an inverter with fan-out four in a given technology are shown in Table 2.

Design	FIFO cycle time	Cell forward latency
RT	9.5	2
SI	11.5	3
RT reshuffled	5.7	2
SI reshuffled	7.6	3

Table 2: Performance comparison of FIFOs normalized to a fan-out four inverter delay

For the first relative timing FIFO (reported in the first row) we use a RT circuit derived by `petrify` using only automatic timing assumptions presented in Figures 6.(e). A proper transistor sizing is required for correct operation of the circuit. No user-defined assumptions on the environment are used. The timing analysis explained in Section 3 has been applied to the specification, and state encoding has been automatically solved as desribed in Section 4.8. With this strategy, only one additional state signal, x, was required as shown in Figure 6.(c)[4]. There are some interesting aspects of this implementation:

- The state signal x is is switching concurrently with other activity in the circuit.This is a result of the state encoding strategy of `petrify` that attempts to increase the concurrency of new state signals until they disappear from the critical paths according to the delay model explained in Section 3.
- The response time of the circuit with regard to the environment is only one event (two inverters), i.e. as soon as an output event is enabled it fires without requiring the firing of any other internal event.

Finally, the implementation of Figure 6.(d) requires some timing constraints to be correct. After applying the method proposed in Section 4, five timing constraints between pairs of concurrent events have been derived that are *sufficient* for the circuit to be correct. They are graphically represented in Figure 6.(e).

The constraints $l_o+ < x-$ and $r_o+ < x-$ are not independent. Since the implementation of x is $x = \overline{l_o + r_o}$, it is always guaranteed that one of them will hold, whereas the other must be ensured. Since l_o+ and r_o+ are enabled simultaneously, these constraints will always hold if the delay of two gates is longer than the delay of one gate. From the rest of constraints, the most stringent is $x- < r_i+$. In the worst case, both r_i+ and $x-$ will be enabled simultaneously by r_o+. In this case, it is required the delay of $x-$ to be shorter than the delay of r_i+ (from the enviroment). In case of a very fast environment, it can be forced by different techniques, e.g. transistor sizing or delay padding for gate x.

For the second FIFO (the second row of the table) we derived a speed-independent circuit using `petrify` in the mode of *automatic concurrency reduction* [5] without constraining I/O concurrency of the cell. Because of concurrency reduction only one state signal was required [4] like in the case of the automatic RT solution. However, the state signal was on a critical cycle and the implementation of *lo* and *ro* contained additional p-transistors, which made the speed-independent circuit 20-30% slower than the RT one.

5.3 RAPPID control circuits

In this section we compare manually optimized RT control circuits used for RAPPID [16, 15] with those derived automatically with `petrify`. For each example, Table 3, reports: `manual` (obtained by applying relative timing manually), `automatic` (obtained automatically by `petrify`

[4]This new specification is not strictly a Petri net, since the arcs from l_o+ and r_o+ to the *OR* place indicate an *or-causality* relation: $x-$ is triggered by the first event to fire, whereas the token produced by the latest event is implicitly consumed. An equivalent Petri Net is a bit more cumbersome and is omitted for simplicity.

circuit	Area SI$_a$	Area SI$_t$	Area TI	Response time SI$_a$	Response time SI$_t$	Response time TI	State signals SI$_a$	State signals SI$_t$	State signals TI
adfast	18	31	13	2.17	1.00	1.00	2	2	0
alloc-outbound	20	23	22	1.50	1.11	1.00	2	2	2
master-read	65	79	45	2.29	1.33	1.29	7	7	3
mmu0	33	47	20	2.31	1.38	1.38	3	3	0
mmu1	25	32	15	1.60	1.12	1.12	2	2	1
mr0	50	51	30	1.60	1.45	1.15	3	3	2
mr1	36	39	20	2.25	1.19	1.19	4	3	0
nak-pa	24	35	24	1.25	1.00	1.00	1	1	1
nowick	18	19	16	1.50	1.17	1.00	1	1	1
ram-read-sbuf	30	26	21	1.10	1.00	1.00	1	1	0
sbuf-ram-write	24	44	24	1.63	1.00	1.00	2	2	1
sbuf-read-ctl	18	21	16	2.00	1.50	1.50	1	1	1
seq3	18	22	18	1.50	1.00	1.00	2	2	2
seq-mix	23	28	24	1.40	1.20	1.10	2	2	2
vmebus	22	33	17	2.29	1.57	1.57	1	1	0
Total	**424**	**530**	**325**	**1.76**	**1.20**	**1.15**	**34**	**33**	**16**

(a)

circuit	Area SI	Area TI
chu133	15	14
chu150	16	14
converta	19	14
ebergen	16	16
half	8	7
hazard	8	8
mslatch	24	20
trimos-send	30	21
var1	18	8
vbe5b	13	12
vbe5c	10	10
vbe6a	28	24
vbe10b	32	26
wrdatab	35	33
Total	**272**	**227**

(b)

Table 1: Experimental results: specifications without CSC (a) and with CSC (b).

Design	Area (# tr.)			Worst case response time			Average case response time		
	m	a	s	m	a	s	m	a	s
FIFO-A	**22**	**22**	46	**3.0**	**3.0**	9.0	**2.5**	**2.5**	5.7
FIFO-B	16	**15**	46	**2.0**	**2.0**	9.0	**2.0**	**2.0**	5.7
Byte-cntr	32	**27**	71	4.0	**3.0**	5.0	3.0	**2.5**	4.1
Tag-unit	**31**	47	112	**4.0**	**4.0**	8.0	4.0	**2.7**	6.9
Summary	**101**	111	275	3.3	**2.9**	7.75	3.0	**2.4**	5.6

Table 3: Comparison for two generic representative examples (fifo) and two control circuits from RAPPID (byte-control, tag-unit). Response time is measured in gate delays, area in transistors. m: manual, a: automatic, s: speed-independent.

and applying relative timing) and `speed-independent` (obtained automatically by `petrify` without concurrency reduction).

From the table it can be deduced that automatic solutions are quite comparable with manually optimized RT designs. The improvement in response time by applying relative timing is about a factor of 2, substantially better than for the examples of Table 1. This is because the designers of these circuits had a stronger interaction with the tool and provided aggressive timing assumptions on the environment that could not be derived automatically.

6 Conclusions

The method for automatic generation of timing assumptions presented in this paper allows the designer to concentrate on defining those timing assumptions that can only be deduced from a detailed knowledge of the environment. The technique for automatic back-annotation of timing constraints relative to a particular RT circuit provides necessary timing information for the down-stream tools. Timing-aware state encoding allows area/delay optimization of RT circuits.

Relative timing presents a "middle-ground" between clocked and asynchronous circuits, and is a fertile area for CAD development. Both burst-mode[14, 17] and speed-independent specifications are at opposite extremes of a more general class of relative timing specifications.

Ackowledgments We would like to thank Shai Rotem, Luciano Lavagno, Alex Kondratyev and Alexandre Yakovlev for their contributions in motivating this work and developing the theory for synthesis with relative timing.

References

[1] 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.

[2] W. S. Coates, J. K. Lexau, I. W. Jones, S. M. Fairbanks, and I. E. Sutherland. A fifo data switch design experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 4–17, 1998.

[3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Taubin, and A. Yakovlev. Lazy transition systems: application to timing optimization of asynchronous circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 324–331, November 1998.

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. A region-based theory for state assignment in speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 16(8):793–812, August 1997.

[5] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Automatic synthesis and optimization of partially specified asynchronous systems. In *DAC*, pages 100–115, June 1999.

[6] 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.

[7] Alain J. Martin. Synthesis of asynchronous VLSI circuits. In J. Straunstrup, editor, *Formal Methods for VLSI Design*, chapter 6, pages 237–283. North-Holland, 1990.

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

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

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

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

[12] Radu Negulescu and Ad Peeters. Verification of speed-dependences in single-rail handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 159–170, 1998.

[13] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96:3–33, 1992.

[14] S.M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, Dept. of Computer Science, 1993.

[15] S. Rotem, K. S. Stevens, R. Ginosar, P. A. Beerel, C. J. Myers, K. Yun, R. Kol, C. Dike, M. Roncken, and B. Agapiev. RAPPID: An asynchronous instruction length decoder. In *Proc. ASYNC*, April 1999.

[16] K. S. Stevens, S. Rotem, and R. Ginosar. Relative timing. In *Proc. ASYNC*, April 1999.

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