Synthesis of asynchronous control circuits with automatically generated relative timing assumptions

Jordi Cortadella, Michael Kishinevsky, Steven M. Burns, Ken Stevens

Abstract
This paper describes a method of synthesis of asynchronous circuits with relative timing. Asynchronous communication utilizes handshaking 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") show 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 reduce the 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 cycle 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 to the tool and becomes timing constraints. Different valid nets 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 techniques 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] $\langle S, E, T, s_{in} \rangle$, where $S$ is a non-empty set of states, $E$ is a set of events, $T \subseteq S \times E \times S$ 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 $s \rightarrow s'$ 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 $v(s)$ the value of signal $a$ in state $s$. An SG is consistent if rising...
domains, (c,d,e) Circuits.

An STG has an associated 

STG: (a) Signal Transition Graph (STG), labeled with the same event 

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

2.2 Signal Transition Graph

A Signal Transition Graph (STG) is a Petri net (PN) 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 ER(a*), is the set of states such that \( s \in \text{ER}(a*) \Rightarrow s'^* \). The quiescent region of a*, denoted by QR(a*), is the set of states such that \( s \in \text{QR}(a*) \Rightarrow \nu_a(s) = 0 \land s \notin \text{ER}(a*) \). Similarly, \( s \in \text{QR}(a-) \Rightarrow \nu_a(s) = 0 \land s \notin \text{ER}(a+) \). In Figure 2.(b), ER(s+) = \{101, 111\} and QR(s-) = \{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, \( \text{EnR}(a*) \), of a signal transition a* is a the set of states in which transition a* is enabled. The firing region, \( \text{FrR}(a*) \), of a signal transition a* is the set of states from which a* can fire, i.e. \( s \in \text{FrR}(a*) \Leftrightarrow \exists s' : s \Rightarrow s' \).

A potentially enabling region, \( \text{PenR} \), 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: \( \text{FrR}(a*) \subseteq \text{EnR}(a*) \subseteq \text{PenR}(a*) \). 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 \( \text{EnR}(a*) \neq \text{FrR}(a*) \). A state graph is called lazy (lazy SG) if at least one transition is lazy.

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 \( \text{PenR}(b*) \) into \( \text{FrR}(a*) \).

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:

\[
\begin{align*}
\text{f}_y(Z) &= \begin{cases} 
1 & \text{if } \exists s \in \text{ER}(a+) \cup \text{QR}(a+) \text{ s.t. } v(s) = Z \\
0 & \text{if } \exists s \in \text{ER}(a-) \cup \text{QR}(a-) \text{ s.t. } v(s) = Z \\
\text{otherwise} & \end{cases}
\end{align*}
\]

\(^{1}\) 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 $(s \in ER(a+ \cup QR(a+))$ or $s' \in ER(a- \cup QR(a-))$. 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, $f_a(101) = 0; f_a(101) = f_a(101) = 1$ 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:

$$f_a(Z) = \begin{cases} 1 & \text{if } 3s \in FR(a+) \cup QR(a+) \ \text{s.t. } v(s) = Z \\ 0 & \text{if } 3s \in FR(a-) \cup QR(a-) \ \text{s.t. } v(s) = Z \\ \text{otherwise} & \end{cases}$$

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), $f_a(101) = \neg f_a(101) = f_a(101) = 1$ 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:

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

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-free 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 $s \to s'$:

$$\{(s \in S_1 - S_2 \Rightarrow s' \in S_1) \land (s \in S_2 \Rightarrow s' \not\in S_1 - S_2)\}$$

Only monotonic covers of FRs can be selected as ERs for hazard-free solutions for logic netlist [3]. If $S_1 = ER(a*)$ and $S_2 = FR(a*)$, then (1) no disabling of $a*$ is possible and (2) there are no transitions from $FR(a*)$ to $ER(a*) - FR(a*)$, 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 net $\{101, 110, 111\}$ is a monotonic cover of $ER(x -)$. However, the set $\{100, 101, 111\}$ is not, since the transition $100 \to 110$ 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 $\exists s_1 \to s_2$ such that $s_1 \in ER(e_2)$ and $s_2 \not\in$ 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 (Concurrent) 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. $ER(e_1) \cap ER(e_2) \neq \emptyset$
2. $\forall s \in ER(e_1) \cap ER(e_2) : (s \to s_1) \in T \land (s \to s_2) \in T \Rightarrow \exists s_3 \in S : (s_1 \to s_3) \in T \land (s_2 \to s_3) \in T$.

Definition 3.3 (Trigger) An event $e_1 \in E$ triggers another event $e_2 \in E$ (denoted by $e_1 \rightarrow e_2$) if $\exists s_1 \to s_2$ such that $s_1 \not\in ER(e_2)$ and $s_2 \in ER(e_2)$.

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 $e_1 \leq e_2$) if $\exists s_1 \to s_2$ such that $s_1 \not\in ER(e_2)$ and $s_2 \in ER(e_1) \cap ER(e_2)$.

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 $e_1 \otimes e_2$) if $\exists s_1 \to s_2$ such that $s_1 \not\in ER(e_2)$ and $s_2 \in ER(e_1) \cap ER(e_2)$.

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 $X = \{e_1, \ldots, e_n\} \subseteq E$. $e$ can be enabled before $X$ (denoted by $e < X)$ if $\exists s_1 \to s_2$ such that $s_1 \not\in ER(e) - ER(X)$, $s_2 \in ER(e) \cap ER(X)$ and $e' \not\in X$, where $ER(X) = ER(e_1) \cup \ldots \cup ER(e_n)$.

Figure 1b depicts the transition system derived from the Petri net of Figure 1a. The following facts can be derived using the definitions above: $(a \parallel b), c \parallel f, c < f, c \otimes e$, etc. Event $d$ cannot be enabled before $(e, f)$, but can be enabled before $(e, f, g)$ since there is a transition $s_0 \to s_1$ such that $s_0 \not\in ER(d) - ER\{e, f, g\}$, $s_1 \not\in \{e, f, g\}$.

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 $[1, +\infty]$
- 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 $\Delta$ 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, $e_1, e_2, e_3 \in E$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Event enabled before another event.</td>
<td>Ordering relations: $(e_1 \parallel e_2) \land (e_1 \prec e_2) \land (e_2 \prec e_1) \land (e_1 \not\prec e_2)$.</td>
</tr>
<tr>
<td>Difference timing assumption: $e_1$ fires before $e_2$</td>
<td>Delay assumptions: one gate short than two gates.</td>
</tr>
<tr>
<td>II. Events simultaneously enabled.</td>
<td>Ordering relations: $(e_1 \parallel e_2) \land (e_1 \equiv e_2) \land (e_2 \equiv e_1)$.</td>
</tr>
<tr>
<td>Difference timing assumption: $e_1$ fires before $e_2$</td>
<td>Delay assumptions: delay of $e_2$ longer than delay of $e_1$.</td>
</tr>
<tr>
<td>III. Event triggered by events simultaneously enabled.</td>
<td>Ordering relations: $(e_1 \parallel e_2) \land (e_1 \equiv e_2) \land (e_2 \equiv e_1) \land (e_1 \not\equiv e_2) \lor (e_2 \not\equiv e_1)$.</td>
</tr>
<tr>
<td>Simultaneity timing assumption: $e_1$ and $e_2$ simultaneous wrt $e_3$.</td>
<td>Delay assumptions: one gate shorter than two gates.</td>
</tr>
<tr>
<td>IV. Early (speculative) enabling for ordered events.</td>
<td>Ordering relations: $e_1 \rightarrow e_2$.</td>
</tr>
<tr>
<td>Early enabling timing assumption: $e_1$ fires before $e_2$ (but $e_2$ can be enabled concurrently with $e_1$).</td>
<td>Delay assumptions: delay of $e_1$ shorter than delay of $e_2$.</td>
</tr>
</tbody>
</table>

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 $(s_2, s_5, s_8, s_12, s_15, s_18, s_21)$ as don't care states for the values of the next state function for signal $d$ in addition to the originally present states $(s_5, s_6, s_9, s_{13}, s_{16}, s_{19}, s_{22})$.

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.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>V. Input not enabled before non-input event.</td>
<td>Ordering relations: $(e_1 \parallel e_2) \land e_2 \not\prec e_1$.</td>
</tr>
<tr>
<td>Difference timing assumption: $e_1$ fires before $e_2$.</td>
<td>Delay assumptions: delay of environment longer than delay of one gate.</td>
</tr>
</tbody>
</table>

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: $(\forall e \in X : e \not\equiv e) \land e \not\equiv X$.

Delay assumptions: $X$ fires before $e$.

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 $s_5$). 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 $z+ < y+$ and $y+ < x-$.

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 $x = z + y$ 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 $z = 0$ 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.
Let us consider the same example under a simultaneity assumption.

4.2 Example 2

Let us consider the same example under a simultaneity assumption \(x^+ \text{ and } y^+ \text{ are simultaneous with respect to } x^-\). Under this assumption state 001 is unreachable and becomes a don't care for all signals. In addition states 101 and 110 become don't cares for signal\(x\), since both belong to the potential \(\text{EnFr}(x^-)\) 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 \(x = y\) 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 \(110 \in \text{EnFr}(x^-) \land 110 \notin \text{Fr}(x^-)\).

Figure 3 shows Karnaugh maps for the next state functions 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 correspond to concurrency reduction (disabling of signal transitions without persistence 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 \(U\) 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 \(U_L\). 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 \(k, k > n\), signals. Then for comparing states one needs to use a homomorphism \(h : B^k \rightarrow B^n\), that given an implementation state hides \((k - n)\) new internal signals and obtains a specification state. Homomorphism, \(h\), is naturally extended to sets of states.

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

1. \(h(U_L) \subseteq U\), 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., \(s_0 \in S\) 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 \(s_0 \Rightarrow h(s'_0) \Rightarrow h(s'_0) \Rightarrow s_0\) in \(S\) such that sequence \(\tau\) contains only events of internal signals, not observable by the environment.
4. No events disappear: If \(ER(S) \neq 0\), then \(FR_L(e) \neq 0 \land h(FR_L(e)) \subseteq E_{R}(e)\)
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 \(U\) be the set of states reachable in the untimed domain of a state graph and \(T \subseteq U\) 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 \(G = \{(x_1) \ldots | a_m(x)\} \) (where \(X\) is the set of signals) be a set of gates implementing the RT circuit, where \(g_a(X)\) denotes the boolean function implemented by the gate of signal \(a\).

Reachable states in the untimed domain

Let us call \(R(G)\) the set of states reachable in the untimed domain for the circuit \(G\). Note that, in general, \(U - R(G) \neq \emptyset\) due to the reduction of concurrency imposed by the circuit, and \(R(G) - U \neq \emptyset\) 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 \(U_0 = R(G) \cap U\). \(U_0\) can be calculated as follows:

1. For each output signal \(a_i\), calculate \(\text{disabled}(a_i) = \{s \in U \mid s \notin \text{EnFr}(a_i) \land s \notin \text{Fr}(a_i)\}\), i.e. states in \(\text{unreachable states}\).
let us call States with incorrect behavior Backannotation: problem formulation
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 \( \text{incorrect}(a_i) = \{ s \in (U - T) \mid s \in \text{EnR}(a_i) \land s \notin \text{QR}(a_i) \} \), i.e., states in which \( a_i \) was enabled in the untimed domain, but enabled in the circuit.

2. \( \text{incorrect}(G) = U \cap \{ U \cup \text{incorrect}(a_i) \} \)

Backannotation: problem formulation

We need a set of constraints that make the states in \( \text{incorrect}(G) \) 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 \( \text{incorrect}(G) \) is the set of states reachable after applying the constraints for backannotation. Initially we have \( U = \{ S_0, \ldots, S_{10} \} \) and \( T = \{ S_0, S_1, S_2, S_4, S_5, S_9, S_{10} \} \). Let us assume that \( S_0 \) and \( S_{10} \) 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. 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 \( T \) and does not intersect the set of incorrect states \( \{ S_0, S_{10} \} \). 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 \( T \). The covering problem can be formulated as follows:

\[
(e < c) \land (b < d \lor b < e)
\]

with the minimum-cost solution \( C = \{ e < c, b < e \} \) and \( \text{R}(C) = \{ S_0, S_1, S_2, S_4, S_5, S_9, S_{10} \} \).

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 \( \text{incorrect}(G) \) and that have not been removed by previous constraints. This process is iteratively repeated until all incorrect states become unreachable.

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.
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, $T$.
- 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, SI, 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, TI, we observe a reduction of area and delay optimisation for speed-independent design, and relative timing results correspondingly.

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

<table>
<thead>
<tr>
<th>Design</th>
<th>FIFO cycle time</th>
<th>Cell forward latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>9.5</td>
<td>2</td>
</tr>
<tr>
<td>SI</td>
<td>11.5</td>
<td>2</td>
</tr>
<tr>
<td>RT shuffled</td>
<td>5.7</td>
<td>2</td>
</tr>
<tr>
<td>SI shuffled</td>
<td>7.6</td>
<td>3</td>
</tr>
</tbody>
</table>

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

FIFO and a forward latency (an average event propagation time from $t_i$ to $r_o$) 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.

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 described in Section 4.8. With this strategy, only one additional state signal, $x$, was required as shown in Figure 6(c). There are some interesting aspects of this implementation:

- The state signal $x$ 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 $l_i+ < x- + r_o + < x-$ are not independent. Since the implementation of $x$ is $x = l_i + r_o$, it is always guaranteed that one of them will hold, whereas the other must be ensured. Since $l_i +$ 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 $x- < x+$. 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 environment). 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 $l_i$ and $r_o$ 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

3 This new specification is not strictly a Petri net, since the arcs from $l_i +$ and $r_o +$ to the OR place indicate an $n$-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.
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.

Acknowledgments

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


