Lazy Transition Systems and Asynchronous Circuit Synthesis With Relative Timing Assumptions

Jordi Cortadella, *Member, IEEE*, Michael Kishinevsky, *Senior Member, IEEE*, Steven M. Burns, Alex Kondratyev, *Senior Member, IEEE*, Luciano Lavagno, *Member, IEEE*, Kenneth S. Stevens, *Senior Member, IEEE*, Alexander Taubin, *Senior Member, IEEE*, and Alexandre Yakovlev, *Member, IEEE*

***Abstract*—This paper presents a design flow for timed asynchronous circuits. It introduces *lazy transitions systems* as a new computational model to represent the timing information required for synthesis. The notion of laziness explicitly distinguishes between the enabling and the firing of an event in a transition system.**

Lazy transition systems can be effectively used to model the behavior of asynchronous circuits in which relative timing assumptions can be made on the occurrence of events. These assumptions can be derived from the information known *a priori* about the delay of the environment and the timing characteristics of the gates that will implement the circuit. The paper presents necessary conditions to generate circuits and a synthesis algorithm that exploits the timing assumptions for optimization. It also proposes a method for back-annotation that derives a set of sufficient timing constraints that guarantee the correctness of the circuit.

***Index Terms*—Asynchronous circuits, lazy transition systems, logic synthesis, relative timing.**

I. Introduction

DURING the last decade, there has been significant progress in developing methods and tools for asynchronous circuit synthesis [1]–[5]. The two chief directions in this work have been the following two synthesis approaches, one based on the Huffman's state machine model [6], [7] and the other deriving from Muller's concept of *speed-independent* circuit [8]. The former, also known as *fundamental mode* circuit design, makes strong assumptions about the delay of the environment compared to that of the circuit. It requires the environment to be slow enough in applying the new input values so as to allow the circuit to stabilize after responding to the previous input. The most well-known method associated with this approach is the one called *burst-mode* (BM) circuit design, developed in [9], [3], and [10]. The second approach, on the contrary, makes no assumptions about the delays of the environment, permitting some of the inputs to switch in response to changes in some of the circuit's outputs, without waiting for their complete stabilization. This model is called *input–output* (IO) mode. The recently developed design methods and software based on signal transition graphs (STGs) [5], [11] exemplify this approach and produce speed-independent circuits, whose behavior is invariant to delays in gates but may be sensitive to wire delays.

The synthesis techniques described in this paper are an attempt to combine the expressive power of STGs (that allow a designer to finely tune concurrency, sequencing and choice) with the optimization power of BM FSMs and manual timing-driven design [12] (that allow a designer to avoid waiting for signals that are known to be stable). By doing so, high optimization levels are achieved, while keeping the flexibility of our CAD framework. Of course, this power comes at a price: our synthesis algorithms are radically more complex than their BM counterparts (but only moderately more so than speed-independent synthesis). Exploration of efficient heuristics to cope with large specifications are left to future work.

A. Incorporation of Timing Information

When trying to incorporate timing information in the synthesis of asynchronous circuits, a chicken–egg problem is posed. On one hand, an efficient synthesis requires knowledge of the temporal behavior *a priori*. However, the actual temporal behavior can only be determined after synthesis, e.g., once the circuit netlist has been defined. This cyclic dependency is typically solved by iterating and converging toward a solution that meets the assumed timing behavior.

The computational model used in this paper is the one of *timed transition systems* [13]. Besides the causal relation among events, a lower ($\delta_{\min}$) and upper ($\delta_{\max}$) bound on the delay of each event is defined. An event can only *fire* δ time units after it has been *enabled*, where $\delta_{\min} \leq \delta \leq \delta_{\max}$. Thus, an explicit distinction between the *enabling* and the *firing* of an event is made.

Fig. 1(a) depicts an event structure that determines a partial order in the firing of a set of events. Delay intervals for each event are also defined. Fig. 1(b) depicts a transition system in which timing information is not considered. Each path represents one possible run of the system. When moving to the model of *timed transition systems*, each event is associated with a time stamp (the firing time) and each state is associated with a time

Manuscript received October 2, 2000; revised April 23, 2001. This work was supported by a grant from Intel Corporation to the University Politècnica de Catalunya, by ESPRIT ACiD-WG Nr. 21949, and by Grant EPSRC GR/M94366. This paper was recommended by Associate Editor L. Stok.

J. Cortadella is with the Department of Software, Universitat Politècnica de Catalunya, Barcelona, Spain.

S. M. Burns, M. Kishinevsky and K. S. Stevens are with the Strategic CAD Lab, Intel Corporation, Hillsboro, OR 79124 USA.

A. Kondratyev is with Cadence Berkeley Labs, Berkeley, CA 94704 USA.

A. Taubin is with Theseus Logic, Sunnyvale, CA 94086 USA.

L. Lavagno is with the Department of Electronics, Politecnico di Torino, Torino, Italy.

A. Yakovlev is with the Department of Computing Science, University of Newcastle Upon Tyne, U.K.

Publisher Item Identifier S 0278-0070(02)01050-3.

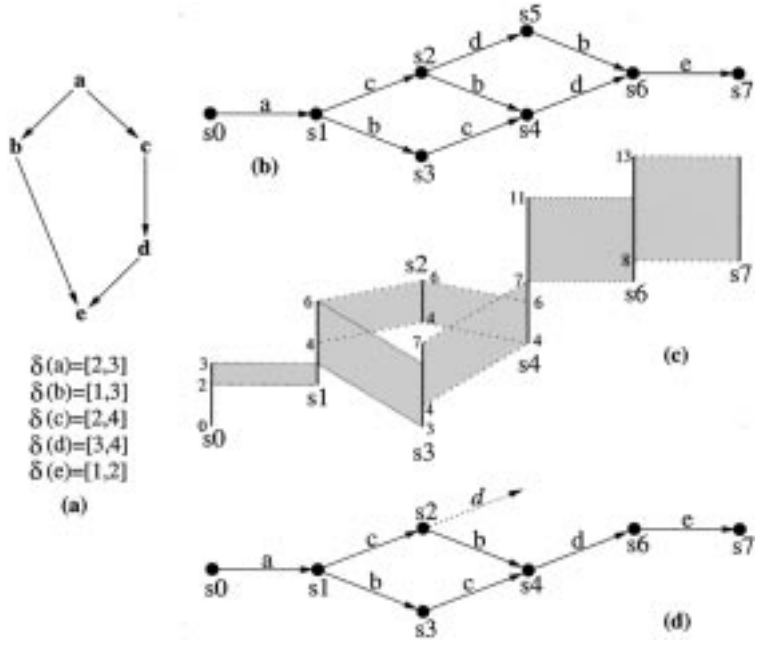


Fig. 1. (a) Event structure with timing information, (b) transition system, (c) timed transition system, and (d) lazy transition system.

interval. Fig. 1(c) is a graphical representation of the state space of the system, starting from time zero. Each vertical edge represents the reachable time stamps that can be associated with any discrete state. For example, in the reachable state space one may find time stamps for s_4 in the interval [4], [11]. Shadowed faces represent state transitions in time-consistent runs of the system. For example

$$s_0 \xrightarrow{a@2} s_1 \xrightarrow{b@4} s_3 \xrightarrow{c@5} s_4 \xrightarrow{d@8} s_6 \xrightarrow{e@10} s_7$$

is a time-consistent run, in which each event is associated with the time stamp of its firing time. However, the run

$$s_0 \xrightarrow{a@2} s_1 \xrightarrow{c@4} s_2 \xrightarrow{d@7} s_5 \xrightarrow{b@8} s_6 \xrightarrow{e@10} s_7$$

is time inconsistent. This can easily be proved by realizing that event b is enabled in the state s_1 at time 2 and fires in state s_5 at time 8, thus being enabled for six time units. However, the delay of event b in the specification is within the interval [1], [3]. The proof that there is no valid run that visits state s_5 can also easily be made, since event b will always fire before event d.

In [14] and [2], *timed circuits* were introduced, also exploiting the fact that timing information can be used to reduce the reachable state space. This helps to eliminate undesired states that do not fulfill implementability properties (e.g., state encoding or persistency) and increase the don't care space during logic minimization. However, it requires the definition of absolute timing information on the delays of the components of the system. While this is possible and useful after at least one design iteration has been completed, it is much more difficult to use at the beginning of the synthesis flow for a variety of reasons.

- Asynchronous specifications are often incomplete and require the addition of state signals, for which no absolute timing information is available.
- Even after state encoding, no absolute timing information about noninput signals of the circuit is known before both technology independent (logic synthesis) and technology dependent (technology mapping) optimizations have been performed. This leads to a chicken and egg problem in any method based on absolute timing information: for efficiency synthesis needs delay bounds, but delay bounds are unknown before synthesis is completed. In timed synthesis this is solved by iterating delay guessing and synthesis.
- All modern synthesis flows both for custom and ASIC design include transistor or gate sizing, buffer insertion, and selection of parameters (e.g., threshold voltage Vt) with the goal of meeting timing constraints and optimizing different design aspects (power, area, delay, etc.) A netlist can be sized differently depending on a given set of constraints, and the resulting gate delays may differ by an order of magnitude depending on the sizes of devices and other selected parameters.
- Placement and routing may further change absolute delay information associated with circuit elements.

Moreover, the formal verification problem with absolute timing becomes drastically more complex due to the need to keep absolute time information, e.g., in the form of regions, in addition to untimed system states [15]. Instead of using absolute delay bounds for the purpose of synthesis, we use relative delay information between circuit events, following the established engineering practice of many high-speed circuit design groups (see e.g., design of pulse-domino logic in [16]). A verification flow following the synthesis flow requires absolute delay information. Different techniques for timing verification can be used, e.g., [2] and [17]–[20] to name a few. Use of relative timing information can be beneficial for verification as well, as shown in [19] and [20].

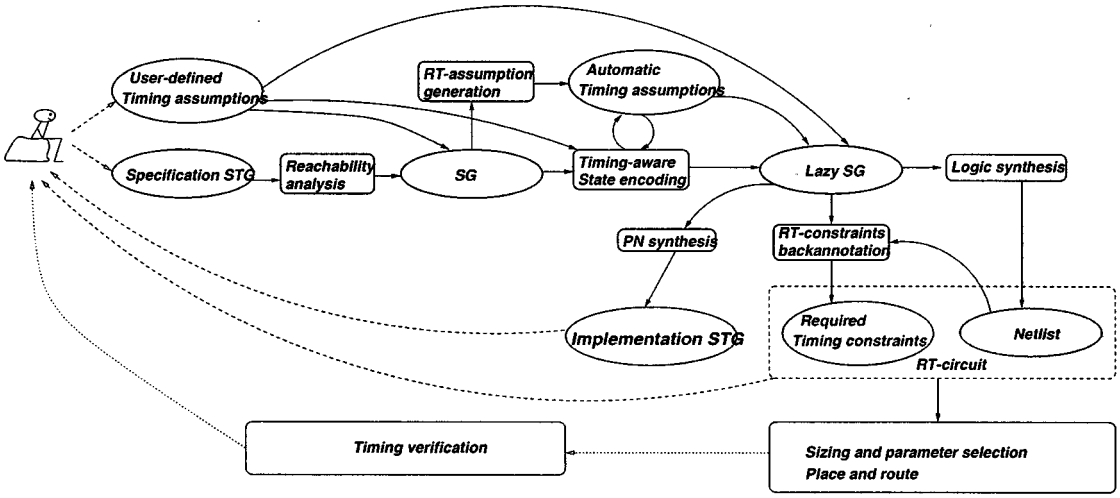


Fig. 2. Design flow for relative timing circuits.

B. Abstraction of Time

Rather than calculating the exact time intervals in which each state can be visited by any valid run, it is sufficient for synthesis to know whether each state is visited by some time-consistent run and what the enabling conditions for every visited state are. In other words, only *the set of reachable states in the timed domain* and the values of next-state function for every signal in every reachable state are needed. This information can be represented by abstracting absolute timing out of the model. This abstraction leads to the definition of a new computational model called a *lazy transition system* [21], in which timing information is only represented by making a distinction between the enabling and the firing of an event.

While absolute timing requires complex techniques to represent the space of reachable timed regions or states (e.g., difference bound matrices, polyhedra, etc.), the generation of the reachable state space for relative timing is of the same complexity as for untimed systems.

Fig. 1(d) represents the lazy transition system associated to Fig. 1(c). The dashed arc with event d from state s_2 indicates that d is enabled in that state, but it cannot actually fire due to its delay. Therefore, state s_5 is unreachable.

This paper proposes a synthesis flow in which timing information is specified as a set of assumptions that relate the firing order of concurrently enabled events, such as event b will always fire before event d. Lazy transition systems are used as the computational model for synthesis.

C. Synthesis Flow

The synthesis flow proposed in this paper follows the paradigm *"assume and, if useful, guarantee."* Similar principles have been used in recent asynchronous designs [12], [22]–[24]. Given an untimed computational model, e.g., a transition system, synthesis of an asynchronous circuit is performed as follows.

1) Derive a set T of timing assumptions on the behavior of the system.
2) Synthesize the circuit by using a subset $T' \subseteq T$ of useful timing assumptions.
3) Derive a set C of sufficient timing constraints that guarantee the correctness of the circuit's behavior.
4) Transistor sizing and parameter selection for a set of constraints C (and possibly some other design constraints).
5) If the set C cannot be guaranteed, calculate a less stringent set T and go to Step 2).

In Step 1), timing assumptions can be either provided by the designer or generated automatically [25]. In the first case, the assumptions typically come from the knowledge of the temporal behavior of the environment, e.g., some of the input events are slow. In the second case, realistic assumptions on the implementation of a circuit can be considered, e.g., the delay of one gate is typically shorter than the delay of two gates.

Not all the timing assumptions in T may be needed to improve the quality of the circuits. During synthesis, only a subset of $T' \subseteq T$ is used for optimization.

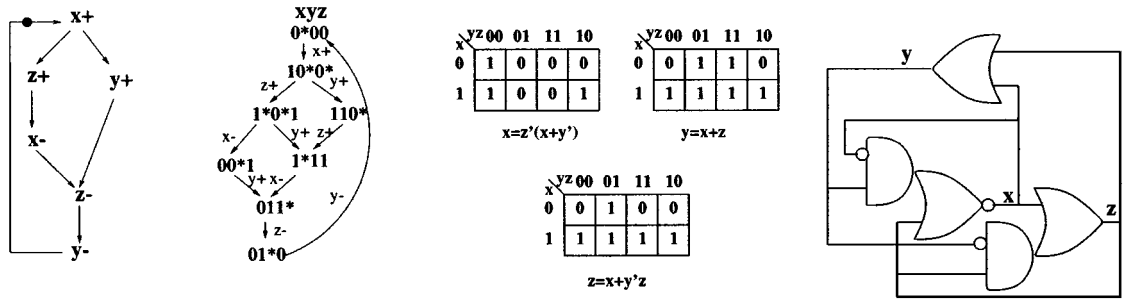
The goal of Step 3) is to find a less restrictive set of constraints that guarantees the circuit's correctness. These constraints may not necessarily match the timing assumptions in T.

Once the circuit and the set C have been derived, the designer must guarantee that the required timing constraints are met. This can be achieved, if necessary, by modifying the actual delays of the components, for example, by delay padding or transistor sizing.

Finally, Step 5) is required to converge in the chicken and egg problem when the initial set T of assumptions results in a circuit that cannot meet the set C of constraints. This design flow is graphically represented in Fig. 2.

The main contributions of this design flow are the following.

- Lazy transition systems are used as a computational model, thus allowing the designer to reason in terms of a partial order of events (relative timing [22]), which is much more intuitive than defining absolute delays when the actual implementation of some components of the system is unknown.
- Timing assumptions can be either provided by the designer or automatically derived from the untimed specification to capture realistic temporal behavior of all "reasonable" implementations.



(a) (b) (c) (d)

Fig. 3. (a) STG, (b) SG, (c) next-state functions, and (d) complex-gate implementation.

- Each circuit is back-annotated with a set of relative timing constraints that guarantee a correct behavior.
- Relative timing allows novel timing optimizations, such as the speculative (early) enabling of events.

It is known [2], [12], [14], [21], [25] that using timing information can significantly improve the quality of synthesized circuits. This paper provides a global formal framework to model, derive, and exploit this information. The synthesis algorithms presented in this paper have been implemented and incorporated in the tool petrify [5].

The paper is organized as follows. Section II presents the computational models used in the paper. Section III presents an overview of the design flow, illustrated with an example. Section IV describes the timing assumptions proposed for circuit optimization in the design flow. The synthesis of circuits from lazy transition systems is discussed in Section V. Next, the strategy used for the automatic generation of timing assumptions is presented in Section VI. The derivation of sufficient timing constraints for correctness is covered in Section VII. Experimental results and conclusions are presented in Sections VIII and IX.

II. Basic Notions

This section presents basic definitions used in the paper. For brevity, the reader is assumed to be familiar with Petri nets, a formalism used to specify concurrent systems. The reader is referred to [26] for a general tutorial on Petri nets.

A. Transition Systems

A *transition system* (TS) is a quadruple [27] $A = (S, E, T, s_{in})$, where S is a *nonempty* set of *states*, E is an alphabet 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 the TS and are often denoted by $s \xrightarrow{e} s'$ instead of (s, e, s'). The notation $s \xrightarrow{e}$ and $\xrightarrow{e} s$ is used when only one of the states of the transition is relevant. Only finite TSs are considered in this paper, i.e., both sets S and T are finite.

The following two definitions are used later in the paper. Given a transition system $A = (S, E, T, s_{in})$, the set of *reachable states* from state s is recursively defined as

$$\text{Reach}(s, T) = \{s\} \cup \bigcup_{s \rightarrow s' \in T} \text{Reach}(s', T).$$

Henceforth, it is assumed that $S = \text{Reach}(s_{in}, T)$ for any TS.

Given a transition system $A = (S, E, T, s_{in})$, and two subsets of states $Y \subseteq X \subseteq S$, the set of states backward reachable within X from Y is defined as

$$\text{BackReach}(X, Y) = Y \cup \bigcup_{s \rightarrow s' \in T, s \in X, s' \in Y} \text{BackReach}(X, \{s\}).$$

In other words, $\text{BackReach}(X, Y)$ are the states in X that have a path within X to some state in Y.

B. State Graphs

In this paper, TSs are used to model asynchronous circuits. For logic synthesis, a binary interpretation of the states and events is required. This interpretation is captured with the notion of a *state graph*.

A *state graph* (SG) is a tuple $G = (A, X, \lambda)$, where $A = (S, E, T, s_{in})$ is a transition system, $X = I \cup O$ is a set of input and output signals and λ is an encoding function. I is the set of signals whose behavior is determined by the environment, whereas O is the set of signals whose behavior is implemented by the system. Therefore, only the signals in O must be synthesized. The set of events E corresponds to rising and falling transitions of the signals, i.e., $E = X \times \{+, -\}$. The symbols $a+$ and $a-$ denote a rising and falling transition of signal a, respectively. The encoding function $\lambda : S \rightarrow \{0, 1\}^n$ assigns a binary vector to each state ($n = |X|$). The code of state s and the value of signal a in s are denoted by $\lambda(s)$ and $\lambda_a(s)$, respectively.

The notation $a*$ is used to denote a transition of signal a in which the fact of rising or falling is not relevant.

An SG is *consistent* if

$$s \xrightarrow{a+} s' \Longrightarrow \lambda_a(s) = 0 \wedge \lambda_a(s') = 1$$
$$s \xrightarrow{a-} s' \Longrightarrow \lambda_a(s) = 1 \wedge \lambda_a(s') = 0$$
$$s \xrightarrow{b*} s' \wedge a \neq b \Longrightarrow \lambda_a(s) = \lambda_a(s').$$

An example of an SG is depicted in Fig. 3(b). The symbol 0^* (1^*) indicates that a rising (falling) transition of the corresponding signal is enabled in that state.

In general, more than one state can be assigned the same code. For simplicity and when no ambiguity is possible, states are often named by their code.

C. Signal Transition Graph

An STG is a Petri net in which transitions are labeled with the same type of events defined for SGs, i.e., rising and falling signal transitions [28], [29].

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

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

Fig. 3(a) depicts an STG with three signals. For simplicity, places with only one input and one output transitions are omitted. Fig. 3(b) shows the corresponding SG with encoded states. The SG is consistent.

D. Circuit Implementation

Given a transition system in which S is the set of states, the *firing region* of an event e, denoted by $\mathrm{FR}(e)$, is the set of states $\{s \mid s \xrightarrow{e}\}$.

The concept of firing region can be trivially extended to SGs. *Quiescent regions* are defined as complements to firing regions

$$\mathrm{FR}(a+) = \{s \mid s \xrightarrow{a+}\};\ \mathrm{QR}(a+) = \{s \mid \lambda_a(s) = 1\} \setminus \mathrm{FR}(a-)$$
$$\mathrm{FR}(a-) = \{s \mid s \xrightarrow{a-}\};\ \mathrm{QR}(a-) = \{s \mid \lambda_a(s) = 0\} \setminus \mathrm{FR}(a+)$$

where " $\setminus$" stands for the set difference.

In Fig. 3(b), $\mathrm{FR}(x-) = \{101,111\}$ and $\mathrm{QR}(x-) = \{001,011,010\}$.

The implementation of an SG as a logic circuit is done by deriving a *next-state function*, $f_a(z)$, for each output signal, a, and binary vector, z. It is defined as follows:

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

E. Implementability Properties

The next-state function f_a of each output signal a is correctly defined when the SG has the *complete state coding* (CSC) property, i.e., when there is no pair of states (s, s') such that $\lambda(s) = \lambda(s')$ and $s \in \mathrm{FR}(a+) \cup \mathrm{QR}(a+)$ and $s' \in \mathrm{FR}(a-) \cup \mathrm{QR}(a-)$. 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.

In the SG of Fig. 3(b), the DC set is empty since all binary vectors have a corresponding state in the SG. As an example, $f(101) = 011$ since signals x and y are enabled in that state. The Karnaugh maps for the next-state functions of signals, x, y, and z are depicted in Fig. 3(c).

Besides consistency and CSC, another property is required for an SG to be implementable as a speed-independent circuit: output persistency [30]. A pair of events (a, b) is *persistent* if for any transition $s \xrightarrow{b} s'$ such that $a \neq b$, $s \in \mathrm{FR}(a) \Rightarrow s' \in \mathrm{FR}(a)$, i.e., a is not disabled by the firing of another event. State s is called nonpersistent if the above condition is violated, i.e., $s \in \mathrm{FR}(a) \wedge s' \notin \mathrm{FR}(a)$

An SG is called *output persistent* if for any pair (a, b) of nonpersistent events, both a and b are events on input signals. In Fig. 3(b), the pair of events $(y+, z+)$ is persistent in the state 100, since the firing of $z+$ leads to the state 101 in which $y+$ is still enabled, and vice versa.

In summary, an SG is implementable as a speed-independent circuit if the following three properties hold: *consistency, complete state coding*, and *output persistency*. In the SG of Fig. 3(b), all the implementability properties for a speed-independent circuit hold.

F. Logic Synthesis

From the next-state functions, a speed-independent circuit can be derived by implementing the Boolean equation of each output signal as an atomic complex gate [8], as shown in Fig. 3(d).

In general, the Boolean equations may be too complex to be implemented as an atomic gate in a specific technology. Methods for logic decomposition and technology mapping that overcome this limitation have been proposed [31], [32]. This paper does not address the problem of technology mapping. However, the proposed optimization methods can be easily combined with existing methods for logic decomposition that can be targeted to technology mapping into given gate libraries.

G. Monotonic Covers

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, S_1 is a *monotonic cover* of S_2 if $S_2 \subseteq S_1$ and for any transition $s \rightarrow s'$

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

Intuitively, once S_1 is entered, the only way to leave it is via a state in its subset ("exit border") S_2.

In the SG of Fig. 3(b), the set $\{101,110,111\}$ is a monotonic cover of $\mathrm{FR}(x-)$. However, the set $\{100,101,111\}$ is not, since the transition $100 \xrightarrow{y+} 110$ violates the conditions for monotonicity.

H. Lazy Transition Systems

A *lazy transition system* (LzTS) is a pair $A = (A', \mathrm{ER})$, where $A' = (S, E, T, s_{in})$ is a transition system and $\mathrm{ER} : E \rightarrow 2^S$ is a function that defines the *enabling region* of each event, in such a way that $\mathrm{FR}(e) \subseteq \mathrm{ER}(e)$ for any $e \in E$. An event e is said to be *lazy* if $\mathrm{ER}(e) \neq \mathrm{FR}(e)$.

The distinction between enabling and firing regions is the abstraction that represents the delay between the enabling of an event and its firing. $\mathrm{ER}(e) \setminus \mathrm{FR}(e)$ is the set of states in which e is enabled but cannot fire. Note that a TS can be considered as a particular case of LzTS in which $\mathrm{ER}(e) = \mathrm{FR}(e)$ for any event.

The binary interpretation of an LzTS is a *lazy state graph* (LzSG) $G = (A, X, \lambda)$, where A is an LzTS and X and λ have the same interpretation as in the previous definition of SG.

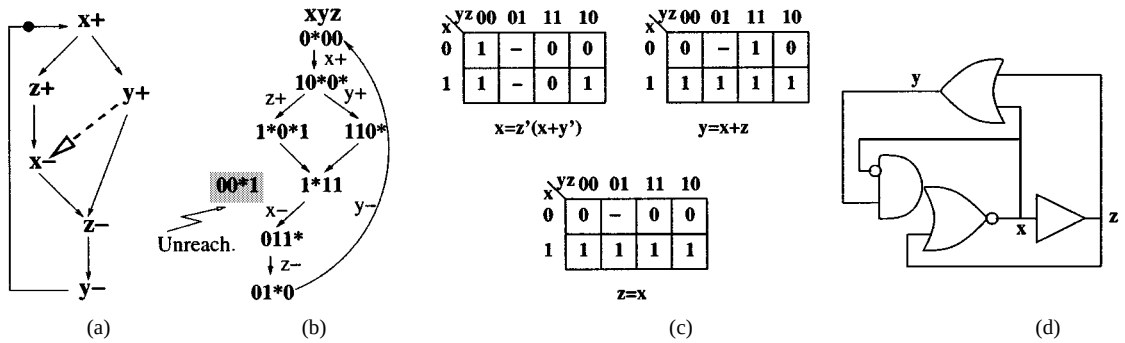


Fig. 4. *xyz* example. Optimization by timed unreachable states.

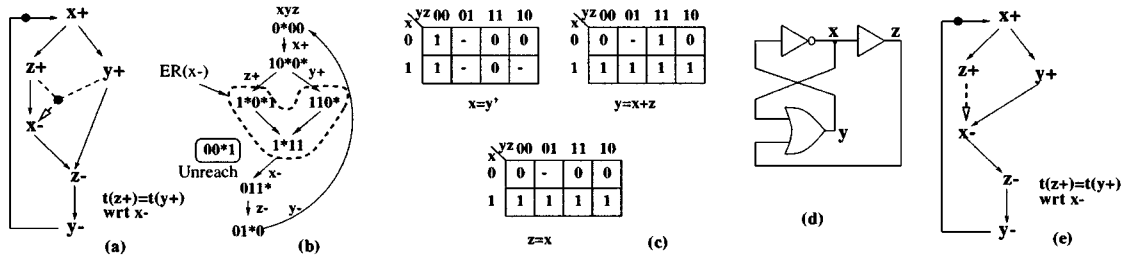


Fig. 5. *xyz* example. Optimization by lazy behavior.

The concept of *lazy quiescent region* (LzQR) is useful for the synthesis of circuits. It is defined as follows:

$$\text{LzQR}(a+) = \text{QR}(a+) \setminus \text{ER}(a-)$$
$$\text{LzQR}(a-) = \text{QR}(a-) \setminus \text{ER}(a+).$$

Synthesis of asynchronous circuits from LzSGs is discussed in Section V.

III. Motivating Example

This section gives an intuitive picture of the optimizations based on timing assumptions. It is illustrated by an implementation of the *xyz* specification shown in Fig. 3(a). This specification describes an autonomous circuit and therefore every signal in the corresponding STG is treated as output. The starting point for optimizations is given by the speed-independent implementation shown in Fig. 3(d).

Speed-independence gives a rather conservative view on gate delays: they are finite but arbitrary. However, more precise timing relationships, considering the time required by a signal to propagate through different stages of logic, can be expressed. For example, one can assume that a signal propagates through a single gate faster than through k gates ($k > 1$), where k is an implementation and/or technology dependent parameter.[1] Similar assumptions were successfully exploited in [33] for area and performance optimization.

Let us assume that the delay of two gates is always longer than the delay of one gate in the circuit for the *xyz* example, using a given technology. Under this assumption, even though the transitions $y+$ and $x-$ are potentially concurrent in the STG, $y+$ would always occur *before* $x-$ in a circuit. In the STG, this timing assumption can be expressed by a special *timing arc* going from $y+$ to $x-$ [34] [denoted by a dashed line in Fig. 4(a)]. Timing restricts possible behaviors of the implementation. In particular, state 001 becomes *unreachable* because it can only be entered when $x-$ fires before $y+$. In unreachable states, the next state logic functions for all signals can be defined arbitrarily [see (1)]. Therefore, the use of timing assumptions increases the DC space for output functions, thus giving extra room for optimization.

For the *xyz* example, moving the state 001 into the DC set of z simplifies its function from $z = x + \overline{y}z$ to a buffer ($z = x$), as shown in Fig. 4(c) and (d). State 101 can be included into the enabling region of $x-$. The selected implementation for signal x, $x = \overline{z}(x + \overline{y})$ is the same as for the untimed specification and corresponds to the $\text{ER}(x-) = \text{FR}(x-) = 111$. Signal x in this implementation is not lazy and no timing constraints are required. An alternative implementation could have been taken with $x = \overline{y} + x\overline{z}$ corresponding to $\text{ER}(x-) = 101{,}111, \text{FR}(x-) = 111$. It might have shorter latency for $x-$ but requires timing constraint $y+$ before $x-$ for correct operation of signal x.

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

In the *xyz* example, $x-$ is the only transition that is affected by $z+$ or $y+$. The dashed hyper-arc from $\langle z+, y+ \rangle$ to $x-$ [see Fig. 5(a)] represents the simultaneity of $y+$ and $z+$ with respect to $x-$. Formally, it means that for the triggering of $x-$, any nonempty subset of the set of events $\{y+, z+\}$ can be chosen. This gives a set of states in which $x-$ can be enabled, $\text{ER}(x-)$, which is shadowed in Fig. 5(b).

[1]This can be formalized in terms of delay range for gates. If a delay range is $[\delta_{\min}, \delta_{\max}]$ then the assumption can be posed as $k * \delta_{\min} > \delta_{\max}$.

It is important to note the following.

- Even though $x-$ might be enabled in any state of ER$(x-)$, its firing (due to timing assumptions) can occur only after $y+$ and $z+$ have fired. This defines FR$(x-) = \{111\}$. This behavior is called lazy because a signal is not eager to fire immediately after its enabling, but waits until some other events have fired.
- Performance can be slightly affected, either positively or negatively, by the fact that the arrival time of the new trigger signals may be different from the ones in the specification.
- The specified ER gives an *upper bound* for the set of states in which a signal can be enabled. In a particular implementation, the actual enabling region can be a subset of the specified enabling region. By exploring different subsets, several implementations can be obtained and evaluated according to some given cost criteria (e.g., area and performance).

The ER of a signal implicitly results in a set of vertices in the DC space of the corresponding logic function. For the enabling of $x-$ in the *xyz* example, different subsets of $\{101,110,111\}$ can be chosen. Transition $x-$ fires at state 111, i.e., FR$(x-) = \{111\}$ and, therefore, any definition of ER$(x-)$ should cover the state 111, since FR$(x-) \subseteq$ ER$(x-)$. Enabling $x-$ in the other two states 101 and 110 can be chosen arbitrarily, i.e., these states can be moved into the DC set of the function for x [see Fig. 5(c)]. After logic minimization, the function for x, which simply becomes an inverter, is defined to be 0 in state 110 and 1 in 101, i.e., the enabling region corresponding to the implementation is ER$(x-) = \{110,111\}$. The back-annotation of this implementation is shown in the STG of Fig. 5(e) in which $x-$ is triggered by $y+$ instead of $z+$. This change of causal dependencies is valid under the assumption that $y+$ and $z+$ are simultaneous with respect to $x-$ and results in $z+$ firing before $x-$. This is indicated by a timing (dashed) arc.

The timed circuit in Fig. 5(d) is much simpler than the speed-independent one in Fig. 3(d). Moreover, if just a single timing constraint *"the delay of $z+$ is less than sum of the delays of $y+$ and $x-$"* is satisfied, then the optimized circuit is a correct implementation of the original specification. Section VII discusses how to derive, from the untimed specification and logic implementation, a *reduced* set of constraints that are sufficient to guarantee its correctness.

Two potential sources of optimizations based on timing assumptions can now be applied:

1) unreachability of some states due to timing (*timed unreachable states*).
2) freedom in choosing enabling regions for signals due to early enabling or simultaneity of transitions (*lazy behavior*).

In both cases, the DC space for the logic functions increases, thus leading to simpler implementations. Unreachable states provide global don't cares (DC for all next state functions), while lazy enabling provides additional local don't cares (DC for the corresponding lazy signal only).

The idea of using the DC space coming from the timed unreachable states is due to [14] and [2] and was successfully exploited in the ATACS tool for the design of timed circuits. To our knowledge, the observation about the additional DC space coming from the lazy behavior appears for the first time in [21] and is the main theoretical contribution of this work. This concept is developed in more detail in the next section.

IV. Timing Assumptions

Timing assumptions could be defined in the form of a partial order in the firing of sets of events, e.g., event a fires before event b. However, this form is ambiguous for cyclic specifications because their transitions can be instantiated many times and different instances may have different ordering relations. More rigor can be achieved at the unfolding level [35], i.e., when the original specification is unfolded into an equivalent acyclic description. The theory of timed unfoldings is however restricted to simple structural classes of STGs and the timing analysis algorithms are computationally expensive [36], [17]. This work relies on a more conservative approximation of timing assumptions in LzTSs.

On the other hand, some specifications explicitly have multiple instances of the same event, e.g., $a+/1$ and $a+/2$, with different causality and concurrency relations. For simplicity in the nomenclature, this paper considers that the same timing assumptions are applied to all instances of the same event. Extending the approach to different assumptions for different instances is quite straightforward.[2]

Some ordering relations between events are first introduced.

Definition 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 $s_1 \in$ ER(e_2) and $s_2 \notin$ ER(e_2). Two events $e_1, e_2 \in E$ are in *conflict* if e_1 disables e_2 or e_2 disables e_1.

Definition 2: (Concurrency) Two events $e_1, e_2 \in E$ are concurrent (denoted by $e_1 \mid\mid e_2$) if:

1) ER$(e_1) \cap$ ER$(e_2) \neq \emptyset$ and they are not in conflict;
2) $\forall s \in$ FR$(e_1) \cap$ FR$(e_2) : (s \xrightarrow{e_1} s_1) \in T \wedge (s \xrightarrow{e_2} s_2) \in T \Rightarrow \exists s_3 \in S : (s_1 \xrightarrow{e_2} s_3) \in T \wedge (s_2 \xrightarrow{e_1} s_3) \in T$.

The second condition is the analogue of the nonconflict requirement but is applied to the FR rather than the ER . It also requires a "diamond" shaped organization of the FR (sometimes called local confluence).

Definition 3: (Trigger) An event $e_1 \in E$ triggers another event $e_2 \in E$ (denoted by $e_1 \rightsquigarrow e_2$) if $\exists s_1 \xrightarrow{e_1} s_2$ such that $s_1 \notin$ ER(e_2) and $s_2 \in$ ER(e_2).

This section proposes three types of timing assumptions. Each assumption enables transformation of an LzTS $A = ((S, E, T, s_{in}), \text{ER})$ into another LzTS $A' = ((S', E, T', s_{in}), \text{ER}')$ in which the set of events and the initial state remain the same, but there is typically more freedom for logic optimization. In A' enabling regions are defined by ER$'$ as *upper bounds* of enabling regions in all

[2]The tool petrify allows us to derive and specify relative timing assumptions for individual instances of the same event.

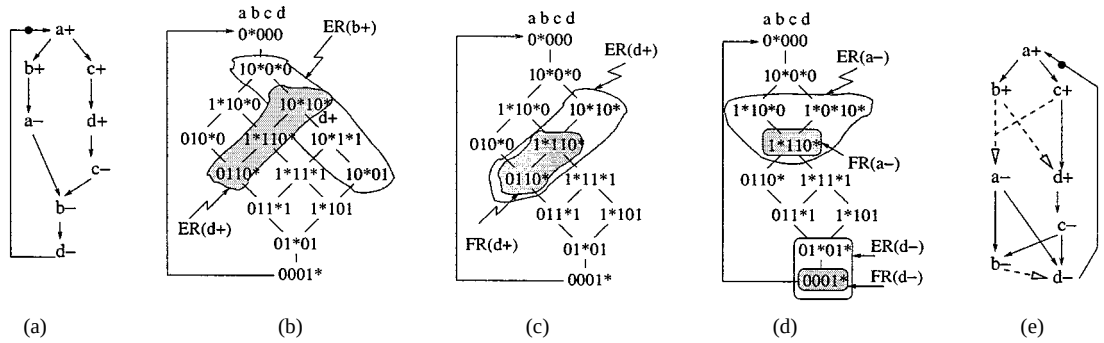


Fig. 6. (a) STG, (b) transition system, (c) LzTS after difference assumption, (d) LzTS after simultaneity and early enabling assumptions, and (e) STG with timing assumptions.

possible implementations according to the considered timing assumption.

A. Difference Assumptions

Given two concurrent events a and b, a *difference assumption* $b < a$ assumes that b fires earlier than a. Formally, it can be defined through the *maximum separation* $Sep_{\max}(b,a)$ between both events [17], [37]. The maximum separation gives an upper bound on the difference between the firing times of b and a. If $Sep_{\max}(b,a) < 0$ then b always fires earlier than a.

In an LzTS, this assumption can be represented by the *concurrency reduction* of a with respect to b. The new LzTS A' is obtained from A as follows.

- Let $C = \mathrm{ER}(a) \cap \mathrm{ER}(b)$.
- $T' = T \setminus \{s \xrightarrow{a} s' | s \in \mathrm{BackReach}(\mathrm{FR}(a), C)\}$.
- $S' = \mathrm{Reach}(s_{in}, T')$.
- For any $e \in E$: $\mathsf{ER}'(e) = \mathsf{ER}(e) \cap S'$.

C is the set of states in which a and b are both enabled (concurrent). The transformation removes the arcs labeled with event a that start in states from C or states from $\mathrm{ER}(a)$ preceding C.

All timing assumptions can be formalized by using the notion of event separation. However, intuition on local timing behavior is enough to reason about the assumptions presented in this paper.

Let us illustrate the application of a difference assumption $b+ < d+$ in the example of Fig. 6(a) and (b). $C = \{1010\}$ and $\mathrm{BackReach}(\mathrm{FR}(d+), C) = \{1010\}$. Thus, the arc $1010 \xrightarrow{d+} 1011$ is removed from T. After that, the set of states $\{1011, 1001\}$ becomes unreachable. The resulting LzSG is depicted in Fig. 6(c) with a lazy event $d+$ in which $\mathrm{FR}(d+) = \{1110, 0110\}$ and $\mathrm{ER}(d+) = \mathrm{FR}(d+) \cup \{1010\}$.

Difference assumptions are the main source for the elimination of timed unreachable states [14], [2], but they cannot fully express the lazy behavior of signals.

B. Simultaneity Assumptions

Simultaneity among a set of events is another kind of timing assumption that has not been exploited explicitly in previous work.[3] It is *relative notion* which is defined on a set of events $E' = \{e_1, \ldots, e_k\}$ with respect to a reference event a, triggered by some of the events in E'. From the point of view of a, the skew in firing times of events in E' is negligible. Formally this can be defined by the following separation inequalities: $\forall e_i, e_j \in E',\ |Sep_{\max}(e_i, e_j)| < \delta_{\min}(a)$, where $\delta_{\min}(a)$ is a lower bound for the delay of event a.

The assumptions are only applicable under the following conditions:

- $\forall e_i, e_j \in E',\ e_i \parallel e_j$;
- $\exists e \in E' : e \rightsquigarrow a$.

Informally, the simultaneity conditions only hold when the events in E' are concurrent and at least one of them triggers a.

The new LzTS A' is obtained from A as follows.

- Let $C = \bigcup_{e_i \in E'} \mathrm{ER}(e_i) \cap \{s | \exists\ s' \xrightarrow{e_j} s : e_j \in E'\}$.
- $T' = T \setminus \{s \xrightarrow{a} s' | s \in \mathrm{BackReach}(\mathrm{FR}(a), C)\}$.
- $S' = \mathrm{Reach}(s_{in}, T')$.
- $\mathrm{ER}'(a) = (\mathrm{ER}(a) \cup C) \cap S'$.
- For any $e \in E, e \neq a$: $\mathrm{ER}'(e) = \mathrm{ER}(e) \cap S'$

C is the set of states in which some event in E' has already fired but some other events in E' are still enabled. Let us consider the simultaneity assumption between transitions $b+$ and $c+$ with respect to $a-$, a being an output signal, in the LzSG from Fig. 6(c). In this case, $C = \{1100, 1010\}$. This assumption influences the LzSG in two ways.

1) State 0100, which is entered when $a-$ fires before $c+$, becomes unreachable. From $|Sep_{\max}(c+, b+)| < \delta_{\min}(a-)$ (coming from the simultaneity assumption) and $Sep_{\max}(b+, a-) < 0$ (coming from the causality between $b+$ and $a-$), the difference assumption $Sep_{\max}(c+, a-) < 0$ can be inferred as well.
2) $\mathrm{ER}(a-)$ is extended to the state 1010 [see Fig. 6(d)].

The second point implies that simultaneity constraints, and hence the possibility of optimization based on them, are inherently more powerful than difference constraints only (that capture only the first point).

C. Early Enabling Assumptions

The simultaneity assumptions exploit "laziness" between concurrent transitions. This idea can be generalized for ordered transitions as well. Assume that event a triggers event b and that the implementation of a is "faster" than that of b (or more formally: $\delta_{\max}(a) < \delta_{\min}(b)$). Then, the enabling of b

[3]Multiple input change in fundamental mode, as defined by Huffman [6], required inputs to change "simultaneously," i.e., within a small time window δ. However, this was not really exploited for optimization, and it did not result in a clean design methodology.

could be started simultaneously with the enabling of a, and the proper ordering of a before b would be ensured by the timing properties of the implementation. In the LzTS this would result in the expansion of ER(b) into ER(a).

Formally, the *early enabling* of event b with respect to a can be applied when $a \rightsquigarrow b$. The new LzTS A' is obtained from A as follows.

- Let $C = \{s | \exists s \xrightarrow{a} s' : s \notin \text{ER}(b) \wedge s' \in \text{ER}(b)\}$.
- $T' = T$.
- $S' = S$.
- $\text{ER}'(b) = \text{ER}(b) \cup C$.
- For any $e \in E, e \neq b : \text{ER}'(e) = \text{ER}(e)$.

The early enabling of $d-$ with respect to $b-$ is illustrated in Fig. 6(d). All of the introduced timing assumptions are shown in the STG of Fig. 6(e), where the dashed arc $(b+, d+)$ corresponds to the difference constraint $b+ < d+$, the hyper-arc $(b+c+, a-)$ corresponds to the simultaneity of $b+, c+$ with respect to $a-$, and the triggering of $d-$ by $a-$ and $c-$ (instead of $b-$) shows the early enabling of $d-$ (the timing arc $(b-, d-)$ is needed to keep the information about the original ordering between $b-$ and $d-$). The transformation for early enabling has been defined only in the case of one backward step, i.e., the implementation of one signal a that triggers b is faster than that of b, and hence b can be enabled at the same time as a and still fire after a purely due to timing. This definition can be generalized for multiple backward steps, i.e., the total delay of the implementations of two signals a and b such that a triggers b and b triggers c is faster than the implementation of c, that can thus be enabled together with a and still fire after b. Of course assumptions going beyond one step are often much less realistic and harder to satisfy.

The above three types of timing assumptions are the cornerstone for timing optimization. Note that difference constraints are mainly used for removal of the timed unreachable states, while simultaneity and early enabling open a new way for simplifying logic by choosing a particularly useful lazy behavior of the signals.

V. Synthesis With Relative Timing

This section presents the theory for the synthesis of hazard-free asynchronous circuits with relative timing assumptions. Lazy transition systems are used as the specification model that incorporates timing.

A. Implementability Properties

The *next-state function* defined for each output signal for the implementation of an LzSG as a circuit is as follows:

$$f_a(z) = \begin{cases} 1, & \text{if } \exists s \in \text{FR}(a+) \cup \text{LzQR}(a+)\ s.t.\ \lambda(s) = z \\ 0, & \text{if } \exists s \in \text{FR}(a-) \cup \text{LzQR}(a-)\ s.t.\ \lambda(s) = z \\ -, & \text{otherwise} \end{cases} \tag{2}$$

Note that this definition generally gives more don't cares than the (1) for SGs due to two reasons.

- More states are unreachable, since timing assumptions can reduce concurrency.
- States in ER $\setminus$ FR do not belong to either FR, or LzQR , and hence are included into the DC-set.

For an LzSG to be implementable as a hazard-free circuit, the properties of CSC and output persistency must be extended.

The CSC property holds in an LzSG when f_a is well defined, that is *if* (but *not only if*) there exists no pair of states (s, s') such that $\lambda(s) = \lambda(s')$ and $s \in \text{ER}(a+) \cup \text{LzQR}(a+)$ and $s' \in \text{ER}(a-) \cup \text{LzQR}(a-)$. The condition can be relaxed because CSC conflicts that involve states from ER $\setminus$ FR could be eliminated by treating ER $\setminus$ FR as a DC-set for f_a. However, in order to simplify things, we treat CSC conflicts only in the framework of the above sufficient condition.

The notion of output persistency (see Section II) can also be extended to LzTSs. If an LzTS is output persistent, then all signals are hazard-free both for the pure and inertial bounded gate delay models [38] when the bounds satisfy the timing assumptions implied by the LzTS.

Definition 4: (Persistency) Given an LzTS $A = (A', \text{ER})$ with $A' = (S, E, T, s_{in})$, an event $e \in E$ is persistent if e is persistent in A' and ER(e) is a monotonic cover of FR(e).

Intuitively, persistency in LzTS indicates that once ER(e) has been entered, it can only be exited by firing e. Moreover, persistency in A' indicates that no transition can switch an event from fireable (in FR(e)) to only enabled (in ER$(e) \setminus$ FR(e)).

Thus, an LzSG is implementable as a hazard-free circuit with pure and bounded delays of its gates if the following properties, extended to LzSGs, hold: consistency, complete state coding, and output persistency. These conditions are an extension to circuits with inputs and relative timing of the semimodularity conditions used by Muller to guarantee hazard-freedom for autonomous circuits with unbounded delays [8], [39], [40].

B. Synthesis Flow With Relative Timing

The flow for logic synthesis with relative timing assumptions is the following.

1) Define a set of timing assumptions on a TS A and derive a specification LzTS $A_T = (A', \text{ER}_T)$ according to the defined assumptions.

   These assumptions must be provided by the designer or generated automatically (e.g., for inserted state signals, as described below). They allow the transformation of the TS in Fig. 3(b) to the LzTS in Fig. 5(b). This paper proposes three types of timing assumptions. They are described in Section IV.
2) The second step of synthesis is state encoding, that is inserting state signals for resolving CSC conflicts and thus making an LzSG implementable. State encoding in our logic synthesis framework is automatically solved using an extension of the method presented in [41].
   - Only those encoding conflicts reachable in the timed domain are considered in the cost function (no effort is invested in solving unreachable conflicts).
   - Timing assumptions can be generated for inserted state signals using the rules from Section VI, implying that the events of state signals can also be lazy.

It is important to notice that the automatic generation of timing assumptions is crucial to optimize the behavior of the circuit when signals not observable in the specification, e.g., signals for state encoding, are considered.

3) Derive another implementation LzTS $A_I = (A', \mathrm{ER}_I)$ in which the implementability conditions hold and $\mathrm{ER}_I(e) \subseteq \mathrm{ER}_T(e)$ for any event e.

   A_T is the LzTS that defines the upper bounds on the ERs of the events, i.e., how early each event can be enabled without firing. A_I defines a particular implementation in which the enabling of each event cannot be earlier than the one defined by A_T. The method for defining A_I from A_T is done through logic minimization and is explained in detail in Section V-C.

4) Derive a circuit implementation for the corresponding LzSG according to the logic functions defined by (2).

5) Back-annotate timing constraints sufficient to the correctness of the implementation.

   Steps 3) and 4) are discussed in Section V. Steps 1) and 5) are presented in Sections VI and 7, respectively. Step 2) is not discussed in more detail, since the basic theory is similar to that for speed-independent circuit synthesis presented in [41].

In the example of Fig. 5(b), the only lazy event is $x-$. For signal x, the following regions are defined:

$$\mathrm{ER}_T(x+) = \{000\}; \quad \mathrm{LzQR}_T(x+) = \{100\}$$
$$\mathrm{ER}_T(x-) = \{101,110,111\}; \quad \mathrm{LzQR}_T(x-) = \{011,010\}.$$

For the circuit in Fig. 5(e), the corresponding A_I fulfills the properties for implementability and has the following regions for signal x:

$$\mathrm{ER}_I(x+) = \{000\}; \quad \mathrm{LzQR}_I(x+) = \{100,101\}$$
$$\mathrm{ER}_I(x-) = \{110,111\}; \quad \mathrm{LzQR}_I(x-) = \{011,010\}.$$

C. Synthesis Algorithm

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

The timing assumptions on the behavior of the circuit and the environment can be specified by the designer or generated automatically (see Section VI). Two types of assumptions are accepted.

- $\tau(a) < \tau(b)$, indicating that event a will occur before event b. In case both events are concurrent, it corresponds to a different assumption. In case a triggers b, it corresponds to the early enabling of b with respect to a.
- $\tau(a) \simeq \tau(b)\ wrt\ c$, indicating that the firing of a and b can be considered simultaneous with regard to c (simultaneity assumption).

In the example of Fig. 5, the following assumptions have been specified for optimization:

$$\tau(y+) < \tau(x-) \text{ and } \tau(y+) \simeq \tau(z+) \text{ wrt } x-.$$

$\mathsf{ON} = \mathsf{FR}(x+) \cup \mathsf{LzQR}_T(x+)$;
$\mathsf{OFF} = \mathsf{FR}(x-) \cup \mathsf{LzQR}_T(x-)$;
repeat
  $\mathsf{C}(x) = \text{Boolean\_minimization}(\lambda(\mathsf{ON}), \lambda(\mathsf{OFF}))$;

  /* Make the ON cover monotonic*/
  $\mathsf{ER}_I(x+) = \lambda^{-1}(\mathsf{C}(x)) \cap \mathsf{ER}_T(x+)$;
  $H_{on} = \{s \in \mathsf{ER}_I(x+) \mid \exists s \rightarrow s' : s' \in \mathsf{ER}_T(x+) \setminus \mathsf{ER}_I(x+)\}$;
  $\mathsf{OFF} = \mathsf{OFF} \cup H_{on}$;

  /* Make the OFF cover monotonic*/
  $\mathsf{ER}_I(x-) = \lambda^{-1}(\overline{\mathsf{C}(x)}) \cap \mathsf{ER}_T(x-)$;
  $H_{off} = \{s \in \mathsf{ER}_I(x-) \mid \exists s \rightarrow s' : s' \in \mathsf{ER}_T(x-) \setminus \mathsf{ER}_I(x-)\}$;
  $\mathsf{ON} = \mathsf{ON} \cup H_{off}$;
until $(H_{on} = \emptyset) \wedge (H_{off} = \emptyset)$;

Fig. 7. Algorithm for logic synthesis of output signal x.

The algorithm for the synthesis of each output signal x is shown in Fig. 7, in which the definition of λ has been extended to sets of states and boolean vectors as follows:

$$\lambda(X) = \{\lambda(s) \mid s \in X\}$$
$$\lambda^{-1}(Y) = \{s \in S \mid \lambda(s) \in Y\}.$$

The algorithm takes an LzTS, A_T, as input and generates another LzTS, A_I, and a logic function $C(x)$ for each output signal, according to the design flow described in Section V-B. In case each function $C(x)$ is implemented as a complex gate, the circuit is guaranteed to be hazard-free under the given timing assumptions.

This heuristic algorithm calculates ER_I iteratively until a monotonic cover is found. Initially, ON and OFF are defined in such a way that the states in $\mathrm{ER}_T(x+) \setminus \mathrm{FR}(x+)$ and $\mathrm{ER}_T(x-) \setminus \mathrm{FR}(x-)$ are not covered, i.e., their binary codes are in the DC set. Boolean minimization is invoked by defining the ON- and the OFF-set, and a completely specified function is obtained. Next, monotonicity of $C(x)$ is checked. H_{on} is the set of states in $\mathrm{ER}_I(x+)$ covered by $C(x)$ that lead to another state in $\mathrm{ER}_I(x+)$ not covered by $C(x)$. These states are removed from $\mathrm{ER}_I(x+)$ for the next iteration. The loop converges monotonically to a valid solution bounded by the case $\mathrm{ER}_I(x+) = \mathrm{FR}(x+)$. A similar procedure is performed on the complement of $C(x)$ for $\mathrm{ER}_I(x-)$. Thus, the DC set is reduced at each iteration of the algorithm to enforce the monotonicity of the cover. This reduction is illustrated in Fig. 8.

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

Petrify includes a Boolean minimizer that delivers several covers with similar cost. One is selected among them by using a prioritized cost function that takes into account monotonicity, literal count, and concurrency. Those covers that include a larger number of states from ER_T are considered to be more concurrent and hence potentially exhibit better global performance.

The algorithm in Fig. 7 generates a netlist of complex gates based on the functions $C(x)$ obtained by the minimization procedure. This algorithm can be easily extended to the synthesis of asynchronous circuits with C elements and Set/Reset functions, $S(x)$ and $R(x)$, corresponding to the enabling of $x+$ and $x-$,

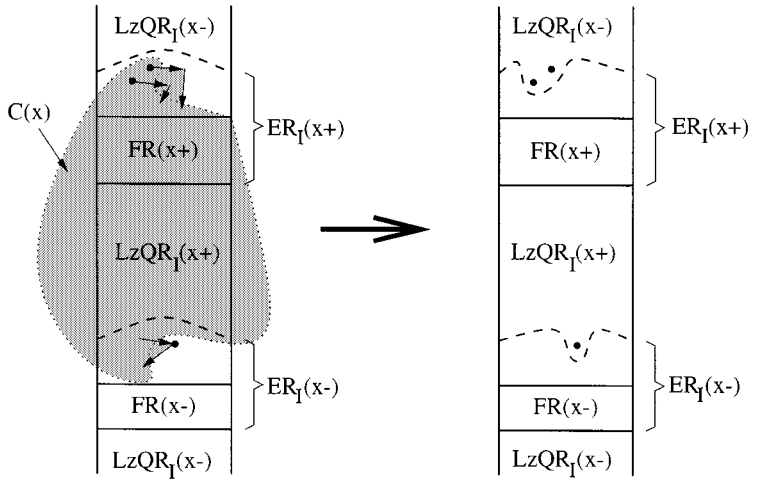


Fig. 8. Iteration to reduce ER_I for nonmonotonic covers.

respectively. The monotonicity conditions for $S(x)$ and $R(x)$ have also been studied in [42] and [43].

VI. Automatic Generation of Relative Timing Assumptions

The timing assumptions described in the previous section can be provided by the designer based on the knowledge the she or he may have of the circuit and its environment. However, many assumptions can be derived automatically by considering some simple delay model, e.g., a unit gate delay model, that may approached to the reality by allowing delay padding or transistor sizing on the synthesized circuit. Here are two typical assumptions that illustrate what can be assumed by the synthesis tool and what must be provided by the designer.

- Synthesis assumption: when two internal signals are enabled simultaneously, one of them can fire before the other. This assumption can be ensured after synthesis by padding some delay to the signal that has been assumed to be slower.
- User-defined assumption: when two inputs are enabled, one will fire before the other. This assumption requires some knowledge about the environment. No assumption can be made *a priori* about the firing order of the events without that knowledge.

The tool petrify enables the designer to provide timing assumptions. These assumptions are checked to be consistently defined according to the behavior of the system, e.g., no difference constraints can be specified between a pair of events that are not concurrent. Moreover, the tool is also capable of generating synthesis assumptions based on a simple delay model. This automatic generation leverages the task of the designer in providing timing information and allows the tool to make assumptions on signals inserted during synthesis and not observable in the specification (e.g., state encoding signals). These assumptions are checked not to contradict any of the user-defined assumptions.

This section presents a method for automatic generation of relative timing assumptions. First, ordering relations between events are defined. Then, the intuition behind this method is explained using a simple delay model for input and noninput events and rules for deriving timing assumptions are given.

A. Ordering Relations

Let $A = ((S, T, E, s_{in}), \mathrm{ER})$ be a lazy transition system.

Definition 5: (Enabled Before) Let $e_1, e_2 \in E$ be two concurrent events. e_1 *can be enabled before*[4] e_2 (denoted by $e_1 \triangleleft e_2$) if $\exists s_1 \rightarrow s_2$ such that $s_1 \in \mathrm{ER}(e_1) \setminus \mathrm{ER}(e_2)$ and $s_2 \in \mathrm{ER}(e_1) \cap \mathrm{ER}(e_2)$.

Definition 6: (Enabled Simultaneously) Let $e_1, e_2 \in E$ be two concurrent events. e_1 and e_2 *can be enabled simultaneously* (denoted by $e_1 \diamondsuit e_2$) if $\exists s_1 \rightarrow s_2$ such that $s_1 \notin \mathrm{ER}(e_1) \cup \mathrm{ER}(e_2)$ and $s_2 \in \mathrm{ER}(e_1) \cap \mathrm{ER}(e_2)$.

The following definition is an extension of definition 5 to sets of events. For a proper understanding, some intuition is required. It is helpful to model the situation in which an event e is much slower than another set of events X and e is never enabled before any of the events in X (see Section VI-C.3). This situation occurs in systems in which the input events (environment) are much slower than the output events. The expected behavior is, thus, that the input event fires after all the output events. The definition itself, however, is concerned with the opposite case, in which an event *can be enabled* before a set of events X, and hence it describes the conditions when timing optimization *cannot be applied*.

Definition 7: (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 $e \triangleleft X$) if $\exists s_1 \xrightarrow{e'} s_2$ such that $s_1 \in \mathrm{ER}(e) \setminus \mathrm{ER}(X)$, $s_2 \in \mathrm{ER}(e) \cap \mathrm{ER}(X)$ and $e' \notin X$, where $\mathrm{ER}(X) = \mathrm{ER}(e_1) \cup \cdots \cup \mathrm{ER}(e_n)$.

In the above definition, e is an event pairwise concurrent with all the events in X. Let us call $\mathrm{ER}(X)$ the union of all excitation regions of the events in X. Since e is concurrent with all events in X, then $\mathrm{ER}(e) \cap \mathrm{ER}(X)$ is not empty.

Now assume that e is a slow event (e.g., from the environment). Assume that internal/output events are very fast (this is a similar situation as in fundamental mode asynchronous circuits

[4]We say "can be" because different occurrences of e_1 can be both before and after e_2. This definition is concerned only with the existence of the former.

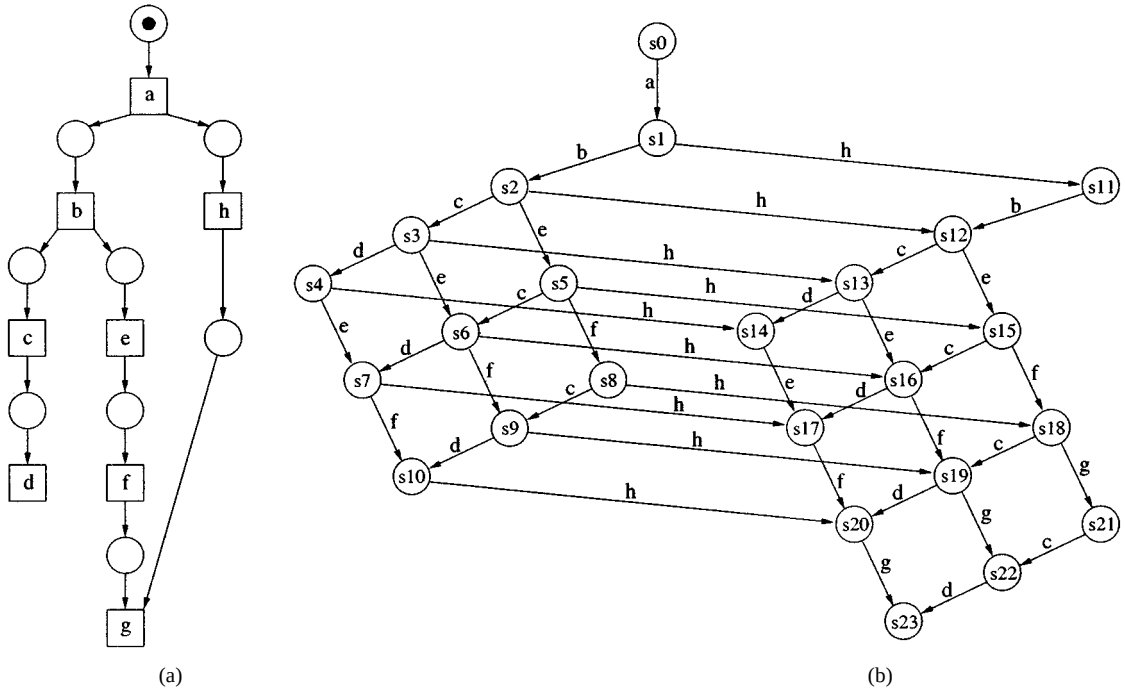


Fig. 9. (a) Petri net and (b) transition system.

[7]). If we know that e is never enabled before entering ER(X), then we know that all events in X will fire before e. This even considers the possibility that the events in X have causality relations among them.

Fig. 9(b) depicts the transition system derived from the Petri net of Fig. 9(a). Events a and b are not concurrent, since ER$(a) = \{s_0\}$ and ER$(b) = \{s_1, s_{11}\}$ are disjoint. Events c and f are concurrent. Moreover, c can be enabled before f since there is a transition $s_2 \rightarrow s_5$ such that $s_2 \in$ ER$(c) \setminus$ ER(f) and $s_5 \in$ ER$(c) \cap$ ER(f). However, f cannot be enabled before c. Events d and f are also concurrent and they can be enabled before each other (see transitions $s_3 \rightarrow s_6$ and $s_5 \rightarrow s_6$). Events c and e are also concurrent but none can be enabled before each other, i.e., they are always enabled simultaneously.

Let us now analyze the enabling relation of event d with some sets of events. 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 $s_9 \in$ ER$(d) \setminus$ ER$(\{e, f, g\})$, $s_{19} \in$ ER$(d) \cap$ ER$(\{e, f, g\})$ and $h \notin \{e, f, g\}$. On the other hand, d cannot be enabled before $\{e, f, g, h\}$.

B. Delay Model

This section presents a *very simple* delay model for events of a TS that gives an intuitive motivation for the automatic generation of timing assumptions. A simple delay model is needed, similar to the literal count in combinational logic synthesis that can be computed *before* deriving a logic implementation and that allows us to bootstrap the timing optimization process. The model, although simple, generates reasonable timing assumptions that can be satisfied by gate selection or transistor sizing. This fact will be shown by comparison with manual designs in Section VIII. This delay model can be changed depending on the design requirements.

The delay of an event is defined as the difference between its enabling time and its firing time. Three types of events are considered[5]:

Noninput events: their delay is in the interval $[1-\varepsilon, 1+\varepsilon]$;
Fast input events: their delay is in the interval $(1+\varepsilon, \infty)$;
Slow input events: their delay is in the interval $[\Delta, \infty)$.

In this context, ε denotes the maximum allowed delay variation of each event with regard to a unit delay. The synthesis approach also assumes that:

- the delay of a gate implementing a noninput event can be increased to be larger than that of another gate by delay padding or transistor sizing;
- the delay of two gates can always be made longer than the delay of one gate. Hence, this imposes the constraint that $\varepsilon < 1/3$;
- 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 assuming a quiescent environment (no input events firing).

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

C. Rules for Deriving Timing Assumptions

Rules for deriving timing assumptions are presented in the following format.

Ordering relations: ordering relations that must be satisfied in an LzTS for a rule to be applied.
Timing assumption: a timing assumption that can be generated automatically.

[5]"Very fast" input events that are not slower than some internal events can be considered as well and treated more or less like noninput events. This consideration is omitted here for simplicity.

Justifying delay assumptions: informal justification of a rule based on the above delay model.

1) Assumptions Between Noninput Events: Assume that $e_1, e_2, e_3 \in E$ are noninput events. The first three rules apply when events e_1 and e_2 are concurrent. The fourth one applies when e_1 triggers e_2. The following rules can be applied for deriving timing assumptions between noninput events:

I) Event enabled before another event.
Ordering relations: $(e_1 \parallel e_2) \wedge (e_1 \triangleleft e_2) \wedge (e_2 \not\triangleleft e_1) \wedge (e_1 \diamondsuit e_2)$.
Difference timing assumption: e_1 fires before e_2
Justifying delay assumptions: the delay of one gate can be made shorter than the delay of two gates.

II) Events simultaneously enabled.
Ordering relations: $(e_1 \parallel e_2) \wedge (e_1 \diamondsuit e_2) \wedge (e_2 \not\triangleleft e_1)$.
Difference timing assumption: e_1 fires before e_2
Justifying delay assumptions: the delay of the gate implementing e_2 can be made longer than the delay of the gate implementing e_1.

III) Event triggered by events simultaneously enabled.
Ordering relations: $(e_1 \parallel e_2) \wedge (e_1 \not\triangleleft e_2) \wedge (e_2 \not\triangleleft e_1) \wedge [(e_1 \rightsquigarrow e_3) \vee (e_2 \rightsquigarrow e_3)]$.
Simultaneity timing assumption: e_1 and e_2 are simultaneous with respect to e_3.
Justifying delay assumptions: the difference in delay of two gates can be made shorter than the delay of one gate.

IV) Early enabling for ordered events.
Ordering relations: $(e_1 \rightsquigarrow e_2)$.
Early enabling timing assumption: e_1 fires before e_2 (but e_2 can be enabled concurrently with e_1).
Justifying delay assumptions: the delay of the gate implementing e_1 can be made shorter than the delay of the gate implementing e_2.

Let us illustrate the previous cases with the example of Fig. 9. Let us assume that all events are noninput. 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). Note that in both cases, the enabling conditions are symmetric, i.e., both events are always enabled simultaneously. However, only one firing order can be chosen by assuming that one of the events can be delayed by increasing the delay of its corresponding gate. This choice can be done heuristically by considering different implementation factors. For example, the choice of one specific firing order may make some states with encoding conflicts unreachable. Another possible heuristic would be to estimate the complexity of the logic for each event. If the gate corresponding to one event is more complex than the other, it can be assumed that the former will be slower than the latter (thus avoiding delay padding to meet the timing assumption).

Timing assumptions of Type III can be applied to the events triggered by the pairs (b, h) and (c, e). Let us analyze the pair (b, h) that triggers the events c, e, and g. The timing assumption informally means that the difference between the firing times of b and h is indistinguishable from the point of view of c, e and g. This opens new possibilities for optimization by using the simultaneity constraints mentioned in Section IV.

Timing assumptions of Type IV can be applied, e.g., to the event d triggered by the event c. For this assumption, the enabling region for d includes the states $\{s_2, s_5, s_8, s_{12}, s_{15}, s_{18}, s_{21}\}$ in addition to the states $\{s_3, s_6, s_9, s_{13}, s_{16}, s_{19}, s_{22}\}$ already in the firing region.

2) Assumptions Between Noninput and Input Events: Assume that $e_1, e_2 \in E$ are a noninput and an input event, respectively, and that they are concurrent.

V) Input not enabled before noninput event.
Ordering relations: $(e_1 \parallel e_2) \wedge e_2 \not\triangleleft e_1$.
Difference timing assumption: e_1 fires before e_2.
Justifying delay assumptions: the delay of environment is longer than the delay of one gate.

This assumption is similar to Types 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 (both slow and fast) will always be longer than the delay of one gate.

3) Assumptions Between Noninput 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 noninput events and e is pairwise concurrent with all the events in X.

VI) Slow input not enabled before noninput events.
Ordering relations: $(\forall e_i \in X : e \parallel e_i) \wedge e \not\triangleleft X$.
Difference timing assumptions: X fires before e.
Justifying delay assumptions: the delay of the slow input event is longer than Δ (the delay required by the circuit to stabilize under a quiescent environment).

To illustrate the meaning of this timing assumption, the example of Fig. 9 is considered, where h is an input event and d is a slow input event. The rest of the events are noninput. After firing the events a, b and c a state in which d, e and h are enabled is reached (s_3). At this point it can be assumed that e and f will fire before d (two gate delays versus 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 the delay can be assumed. If h had been a noninput event, d would be assumed to fire after h and g also.

VII. Back-Annotation of Timing Constraints

Logic synthesis with relative timing assumptions is able to derive a hazard-free circuit that is correct in the timed domain, i.e., in that subset of states of the untimed domain that is reachable by applying the timing assumptions. After the logic synthesis step the assumptions contributing to the synthesis results are propagated to the back-end (e.g., sizing) tools as a set of constraints to be satisfied. After back-end design is completed the validity of the timing constraints must be verified or validated to ensure the correct function of the circuit.

Some of the timing assumptions provided by the user or automatically generated do not contribute to restricting the set of reachable states or the set of transitions and hence are redundant. Moreover, the circuit netlist derived by logic synthesis may

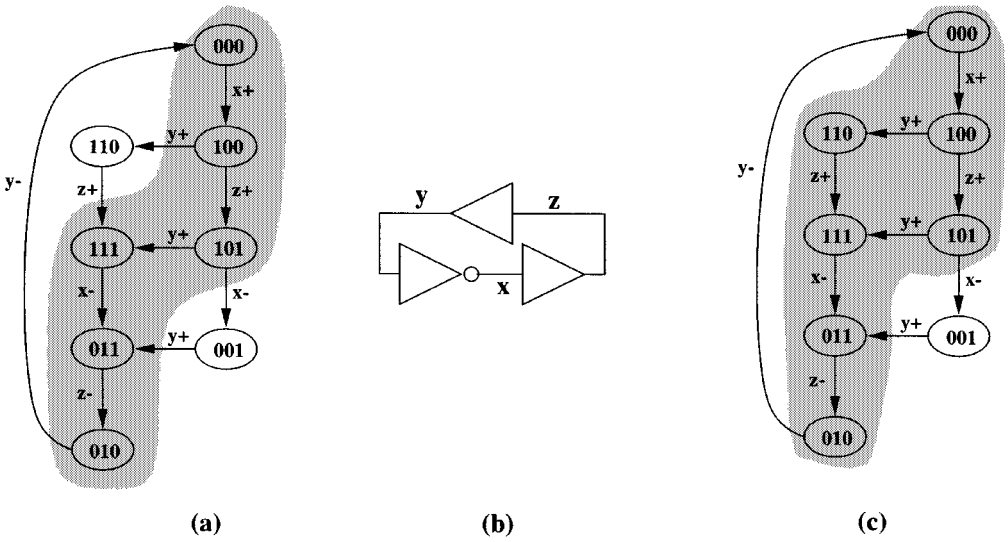


Fig. 10. (a), (c) SGs with timed domains. (b) Circuit.

be correct for a set of states *larger* than the one defined by the timed domain, i.e., one 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 back-annotated 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 are a part of the implementation, since they constitute *sufficient* requirements to be met for a particular netlist solution to be valid.

A. Example 1

Let us analyze the example in Fig. 10. The shadowed states in the SG of Fig. 10(a) correspond to the timed domain determined by the timing assumptions

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

Under these assumptions, logic synthesis can be performed by considering the states 110 and 001 unreachable.

The circuits of Figs. 4(d) and 10(b) have a correct behavior under the stated assumptions. Looking at the circuit of Fig. 4(d) the following can be observed.

- 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, $z-$ is enabled according to the next state function of the implementation, but it is not enabled according to the specification.

Thus, even though the circuit has been obtained using the DC set implied by both assumptions, only one relative timing constraint $y+ < x-$ must be ensured for the circuit to be correct, because *only part of the enlarged DC set has been used in a way that is inconsistent with the original specification*. 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.

B. Example 2

Let us now take the implementation of Fig. 10(b) and analyze the gate $x = \overline{y}$, while ignoring the other gates for now. With regard to the untimed domain, the next-state function for x disagrees with the gate $x = \overline{y}$ in three states: 001, 110, and 101. But the consequences are different in each state. In 110, x should remain stable at 1. However, the gate $x = \overline{y}$ makes the transition $x-$ enabled in state 110. To preserve circuit correctness two options are possible.

1) The state 110 could be made unreachable by concurrency reduction. This in turn could be achieved in two ways:
   - by concurrency reduction in the untimed domain, based on changing logic (i.e., trigger) dependencies between signals as described in [44], [45];
   - by concurrency reduction in the timed domain, based on relative timing constraints that would preserve concurrency for *enabling*, but restrict concurrency for *firing* of signal transitions.
2) The state 110 could remain reachable, while $x-$ would be enabled but not fireable, since another enabled transition fires before $x-$. More formally: $110 \in \text{ER}(x-)/\text{FR}(x-)$.

Similar considerations can be made for state 001.

State 101 illustrates a different case. According to the original specification SG, $x-$ is enabled in 101. In the implementation, however, signal x is stable in 101. This corresponds to a concurrency reduction for signal x in the untimed domain, and this is generally considered to be a valid implementation of the original specification. Concurrency is reduced because state 101 becomes a don't care vector for signal x when 001 is assumed to be unreachable (see Section IV). In summary, for the correctness of the gate $x = \overline{y}$, it is sufficient that the states 110 and 001 are unreachable. However, the gate $x = \overline{y}$ ensures that state 001 is unreachable. Hence only 110 must be made unreachable

TABLE I
CORRECTNESS REQUIREMENTS FOR THE CIRCUIT OF FIG. 10(B)

Gate	Unreachable states required	Unreachable states ensured by logic
$x = \overline{y}$	110, 001	001
$y = z$	110	110
$z = x$	001	

by timing constraints or by further concurrency reduction at the logic level.

A similar analysis can be done for the gates $y = z$ and $z = x$. The sufficient requirements for the correctness of all three gates are summarized in Table I. Interestingly, it can be concluded that the circuit is correct under any timing assumption, i.e., it is speed-independent, since all states required to be unreachable are forced to be unreachable by the concurrency reduction due to the chosen gate implementation. In particular, state 001 needs to be unreachable for gate $z = x$ to be a correct implementation of signal z and it is made unreachable by implementing signal x with a gate $x = \overline{y}$

C. Example 3

Let us consider the same example under the assumption " $z+$ and $y+$ are simultaneous with respect to $x-$." Under this assumption, state 001 is unreachable. In addition, states 101 and 110 become don't cares for signal x, since both belong to ER($x-$) according to the semantics of the simultaneity assumption.

Only one timing constraint, $z+ < x-$, is sufficient for the circuit in Fig. 5(d) to be correct. Gate $x = \overline{y}$ is not enabled in 101, hence concurrency is reduced in this state with respect to the original specification and state 001 becomes unreachable under any gate delay. On the contrary, state 110 corresponds to the expansion of ER($x-$). This enabling is lazy since $110 \in$ ER($x-$)/FR($x-$).

D. Correctness Conditions

The synthesis flow presented in this paper starts with an untimed specification $A = (S, E, T, s_{in})$. After logic synthesis with timing assumptions, a gate implementation is obtained.

Let us consider the circuit operation, ignoring timing assumptions. The untimed behavior of the gate implementation from a given initial state s_{in} can be represented by a transition system $A_G = (S_G, E_G, T_G, s_{in})$. A_G is obtained from A by substituting T with the new transition relation T_G, that coincides with T for the input events and models the behavior of the gates for the output events. Finally, T_G and S_G are calculated by only considering the reachability set from s_{in}.[6]

In the remainder of this section the following assumptions are used entirely for the sake of simplicity of exposition. They are not the constraints of the theory or the implementation.

- The set of signals of A and A_G are assumed to be the same and the states are assumed to be uniquely identified by their encoding.
- The set of states S_G reachable by circuit G in the untimed domain can be much bigger than the original set S due to the possibility of reaching incorrect corners of behavior. It is sufficient to calculate only a border of incorrect behaviors instead of the entire S_G.
- The original transition system A is not required to be untimed. It can include some timing assumptions (e.g., user-defined timing assumptions regarding the behavior of the environment). This helps to reduce the state space of the original specification for large circuits.

Since A_G is an untimed behavior, T_G may contain transitions not present in T, e.g., those transitions reachable when the timing assumptions used for synthesis are not considered for calculating the reachability space. On the other hand, some transitions in T may not belong to T_G due to the concurrency reduction imposed by the implementation.

The problem to be solved is to find a set of timing constraints such that, after being applied to A_G, a new lazy (timed) transition system $A_C = ((S_C, E, T_C, s_{in}), \mathrm{ER}_C)$ is obtained in such a way that $T_C \subseteq T \cap T_G$ and the gate netlist derived for A_G is still a valid implementation for A_C.[7]

Here, *valid* implementation should satisfy three conditions.

1) The sequences of signal transitions produced by the circuit, when operated within an originally specified environment and timing constraints, are a *subset* of the sequences allowed by the STG (no new transitions is allowed).
2) No new[8] deadlocks (states in which no signal transition is enabled) are created.
3) The implementation is hazard-free, i.e., A_C is output persistent.

Let us define three predicates characterizing the above conditions:

$$\text{new\_tr}(G) = \{s \xrightarrow{a_i *} s' \in T_G \mid s \xrightarrow{a_i *} s' \notin T\}.$$

These are transitions that can fire in A_G (untimed circuit) but cannot fire in the original specification.

Due to the concurrency reduction that might have been applied during logic synthesis, some states of S may become unreachable in S_G. The concurrency reduction eliminates some transitions in T_G that might result in new deadlock states if all outgoing transitions from a reachable state are removed. Such deadlocks can be avoided by making them unreachable during legal circuit operation. Thus, we define

$$to\_deadlock(G) = \{s \xrightarrow{a_i *} \in T_G \mid s' \text{ is a deadlock in } A_G \text{ but not in } A\}.$$

New hazardous states are captured with the following predicate:

$$\text{to\_hazards}(G) = \{s \xrightarrow{a_i *} s' \in T_G \mid s' \text{ is output nonpersistent in } A_G, \text{but not in } A\}.$$

[6]Obviously, circuit operation within A_G, may not be correct outside timing domain, e.g., it may be hazardous.

[7]The set of states is implicitly induced by the initial state and the transition relation.

[8]One may argue that the original STG should not have contained any deadlocks any way, but we do not make such an assumption in the following, i.e., deadlocking specifications are considered legal, and we just do not introduce new ones.

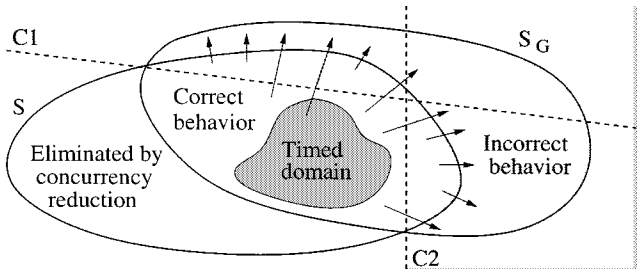


Fig. 11. Formulation of the back-annotation problem. $\{C_1, C_2\}$ is a set of timing constraints sufficient for the correctness of the circuit.

Finally, we define

$$\text{valid}(G) = T_G \setminus (\text{new\_tr}(G) \cup \text{to\_deadlock}(G) \cup \text{to\_hazards}(G)).$$

E. Problem Formulation

The problem to be solved consists of finding a set C of timing constraints, not more stringent than the ones used for synthesis, such that the set of transitions T_C obtained after applying the constraints is a subset of valid(G).

A trivial solution to this problem is to take the complete set of timing assumptions used for logic synthesis. Our goal, however, is to find a less stringent set of constraints sufficient to make the circuit correct. In general, we should look for such a set of constraints that "makes most sense" or that is easiest to satisfy. But the solution of this optimization problem, unfortunately characterized by a very fuzzy cost function, is left to future work.

Instead, a state-based cost function is used to guide heuristics aiming at finding the set C of timing constraints. The cost function is based on the following observation: large state spaces generally require simple constraints.

A corner case of the back-annotation problem would be the situation in which a speed-independent circuit is derived after synthesis with timing assumptions. In that case, the solution to the problem would be an empty set of timing constraints (see Example 2 in this section).

Fig. 11 illustrates the back-annotation problem. The arrows denote the invalid transitions of the circuit. The "timed domain" represents that state space of the circuit under all timing assumptions. $S_G \cap S$ represents the state space in which the circuit behaves correctly. Similarly for the transitions not exiting $S_G \cap S$. The constraints C_1 and C_2 are less stringent than the timed domain defined by all timing assumptions and are enough to guarantee the correctness of the circuit. Note that the states in $S \setminus S_G$ are those eliminated by concurrency reduction. Also note that constraint $C1$ cuts one of the transitions from the timing domain to the region of incorrect behavior, which otherwise might occur due to early enabling.

F. Finding a Set of Timing Constraints

Relative timing constraints are defined in terms of firing order of events. Constraining the firing order between a pair of events only makes sense when they are concurrently enabled. 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 $C_i = (e_j < e_k)$, the set of arcs $disabled(C_i)$ are defined as

$$\begin{aligned} disabled(C_i) = &\{s \xrightarrow{e_k} s' \mid \exists s \rightarrow s_1 \rightarrow \cdots \rightarrow s_n : \\ &s_1, \ldots, s_{n-1} \in \text{FR}(e_k) \wedge s_n \\ &\in \text{FR}(e_k) \cap \text{FR}(e_j)\}. \end{aligned}$$

In particular, the path $s_1 \rightarrow \cdots \rightarrow s_n$ can be empty if $s \in \text{FR}(e_j) \cap \text{FR}(e_k)$. $disabled(C_i)$ 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 $\text{FR}(e_j) \cap \text{FR}(e_k)$ inside $\text{FR}(e_k)$.

Given a set of constraints $C = \{C_1, \ldots, C_p\}$, $disabled(C_i)$ can be used to compute T_C that is the set of reachable transitions after removing the ones in

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

Finding a set C that removes all transitions not in $valid(G)$ can be posed as a covering problem in which all possible firing order constraints of pairs of events are the covering elements.

Currently, petrify uses a greedy approach to solve the covering problem. It merely consists of choosing the constraint that removes the maximum number transitions not in $valid(G)$ and that have not been removed by previous constraints. This process is repeated until all reachable transitions become valid.

G. Example 4

Fig. 12 shows an example with a simplified version of the back-annotation problem, given that the removed objects are states instead of transitions. Assume that the set of states $S_G =$

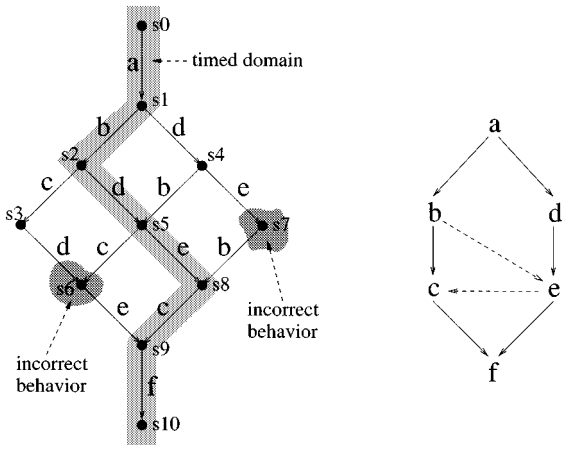


Fig. 12. Example for back-annotation.

TABLE II
UNREACHABLE STATES FOR EACH PAIR OF ORDERED EVENTS $e_1 < e_2$ IN THE EXAMPLE OF FIG. 12. THE PAIRS IN BOLD INDICATE THOSE CONSTRAINTS THAT PRESERVE THE TIMED DOMAIN

$<$	b	c	d	e
b			$\{\mathbf{s_4}, \mathbf{s_7}\}$	$\{\mathbf{s_7}\}$
c			$\{s_4, s_5, s_7, s_8\}$	$\{s_7, s_8\}$
d	$\{s_2, s_3\}$	$\{\mathbf{s_3}\}$		
e	$\{s_2, s_3, s_5, s_6\}$	$\{\mathbf{s_3}, \mathbf{s_6}\}$		

TABLE III
EXPERIMENTAL RESULTS: SPECIFICATIONS WITHOUT CSC (A) AND WITH CSC (B)

	Area			Response time			State signals				Area	
circuit	$\mathbf{SI}_a$	$\mathbf{SI}_t$	**TI**	$\mathbf{SI}_a$	$\mathbf{SI}_t$	**TI**	$\mathbf{SI}_a$	$\mathbf{SI}_t$	**TI**	**circuit**	**SI**	**TI**
adfast	18	31	13	2.17	1.00	1.00	2	2	0	chu133	15	14
alloc-outbound	20	23	22	1.50	1.11	1.00	2	2	2	chu150	16	14
master-read	65	79	45	2.29	1.33	1.29	7	7	3	converta	19	14
mmu0	33	47	20	2.31	1.38	1.38	3	3	0	ebergen	16	16
mmu1	25	32	15	1.60	1.12	1.12	2	2	1	half	8	7
mr0	50	51	30	1.60	1.45	1.15	3	3	2	hazard	8	8
mr1	36	39	20	2.25	1.19	1.19	4	3	0	mslatch	24	20
nak-pa	24	35	24	1.25	1.00	1.00	1	1	1	trimos-send	30	21
nowick	18	19	16	1.50	1.17	1.00	1	1	1	var1	18	8
ram-read-sbuf	30	26	21	1.10	1.00	1.00	1	1	0	vbe5b	13	12
sbuf-ram-write	24	44	24	1.63	1.00	1.00	2	2	1	vbe5c	10	10
sbuf-read-ctl	18	21	16	2.00	1.50	1.50	1	1	1	vbe6a	28	24
seq3	18	22	18	1.50	1.00	1.00	2	2	2	vbe10b	32	26
seq-mix	23	28	24	1.40	1.20	1.00	2	2	2	wrdatab	35	33
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**	**Total**	**272**	**227**

(a) (b)

$\{s_0, \ldots, s_{10}\}$ is reachable by the untimed implementation of the circuit and that the set of states $\{s_0, s_1, s_2, s_5, s_8, s_9, s_{10}\}$ is the one reachable after considering the delays of the circuit. However, incorrect behavior is only manifested in the states s_6 and s_7. Table II contains the set of states that become unreachable by reducing the concurrency between each pair of concurrent events.[9] 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 *small* set of ordering constraints between pairs of events such that the new set of reachable states 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 a set of timing constraints that makes a small number of correct states unreachable and keeps the TS strongly connected. Larger sets of reachable states heuristically result in less stringent sets of constraints, thus simplifying the validation or verification of the circuit. Moreover, they often imply more concurrency and hence heuristically result in better global performance.

The problem can be posed as a *covering problem*. The cells of Table II in bold correspond to those constraints that do not remove any state from the timed domain. The covering problem can be formulated as follows:

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

The constraint $e < c$ is the simplest one removing the state s_6. Any other one (e.g., $e < b$) is more stringent. The constraints $b < d$ and $b < e$ are the ones that can remove the state s_7. The minimum-cost solution is

$$C = \{e < c, b < e\}$$

and

$$S_C = \{s_0, s_1, s_2, s_4, s_5, s_8, s_9, s_{10}\}.$$

[9]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.*

VIII. EXPERIMENTAL RESULTS

The techniques for automatic derivation of relative timing assumptions and synthesis of asynchronous circuits using lazy transition systems have been implemented in the tool petrify and applied to control circuits from RAPPID [12] and a set of other benchmarks. First, results for a standard set of academic benchmarks using conservative (unfavorable for RT) performance estimates are shown. Then a detailed analysis of a FIFO example is presented for estimating the real advantages in performance offered by RT, with automatic timing assumptions versus a speed-independent solution with concurrency reduction. Finally, a comparison of RT solutions derived automatically versus manual solutions is presented.

A. Academic Examples

The results for a well-known set of academic benchmarks are presented in Table III.Table III(a) and (b) present the results for specifications without and with *state coding conflicts* respectively.

The experiments have been performed as follows.

- Columns labeled with $\mathbf{SI}_a$ report results for speed-independent circuits derived by inserting state signals with the aim of minimizing area.
- Columns labeled with $\mathbf{SI}_t$ are derived similarly, but with the aim of minimizing delay. Petrify tries to increase the concurrency of the newly inserted signals until they are outside the critical path of the specification. In case the original specification has no encoding conflicts (Table III(b)), there is no difference between $\mathbf{SI}_a$ and $\mathbf{SI}_t$.
- Columns labeled with **TI** report results for RT circuits. Relative timing assumptions are derived automatically by considering the environment to be *slow*. State signals are inserted aiming at delay minimization.

For each experiment, area is estimated as the number of literals of the *set* and *reset* networks of generalized C-elements.

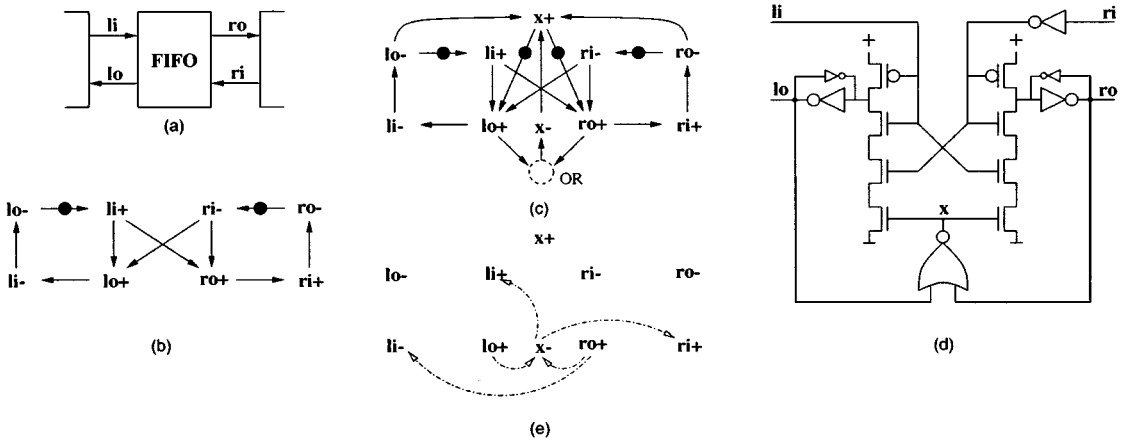


Fig. 13. (a) FIFO controller, (b) specification, (c) specification with state encoding signal, (d) RT implementation with gC elements, and (e) timing constraints sufficient for correctness.

Delay (response time) is estimated as the average number of noninput events in the critical path between the firing of two input events. Given that the estimated response time of the specification does not change when no new signals are inserted, it is not reported in Table III(b).

Relative timing assumptions have a crucial impact on solving state encoding, since petrify inserts new signals only to disambiguate conflicts in the timed domain. Reducing the number of signals also contributes to improving the area and the performance of the circuit.

Comparing the columns $\mathbf{SI}_t$ and **TI**, a reduction of about 40% in area can be observed. The reduction in response time is less than 5% if all events have a delay of one time unit. However, the performance improvement is much more significant if it is evaluated with actual delays, given that the logic of the timed implementation is much simpler. This analysis is reported in Section VIII-B. The improvement obtained for specifications with complete state coding is about 17% in area. This reduction also contributes to improving the performance of the circuits. All the obtained circuits and the corresponding timing constraints were validated by simulation. Only in some cases, transistor sizing or delay padding was required to meet some stringent constraints.

B. Example: A FIFO Controller

This section describes the development of a first-in/first-out (FIFO) cell [specified in Fig. 13(a) and (b)], a simplified abstraction of a part of the RAPPID design. The goal of the specification is to keep the left and right handshakes as decoupled as possible. The modules at the left and right sides of the controller have a similar speed to 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.

Four FIFOs were simulated by using different implementations. The cycle time of the cell was measured. The results, normalized to the delay of an inverter with fan-out of four in a given technology, are shown in Table IV.

The first relative timing FIFO (first row) is an RT circuit derived by petrify using only automatic timing assumptions. It is depicted in Fig. 13(d). 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 VI has been applied to the specification, and state encoding has been automatically solved as described in Section V-B. With this strategy, only one additional state signal, x, was required as shown in Fig. 13(c).[10] There are some interesting aspects of this implementation.

TABLE IV
CYCLE TIME CCOMPARISON OF FIFOS NORMALIZED TO THE DELAY OF AN INVERTER WITH A FAN-OUT OF FOUR

Design	FIFO cycle time
RT	9.5
SI	11.5
RT reshuffled	5.7
SI reshuffled	7.6

- The state signal x is concurrent with other activities 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.
- 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.
- Given that x is never triggering any output signal, the gates of l_o and r_o can be designed by having input x near *Vss*, thus improving their performance.

Finally, the implementation of Fig. 13(d) requires some timing constraints to be correct. Application of the method proposed in Section VII derives five timing constraints between pairs of concurrent events that are *sufficient* for the circuit to be correct. They are graphically represented in Fig. 13(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

[10]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.*

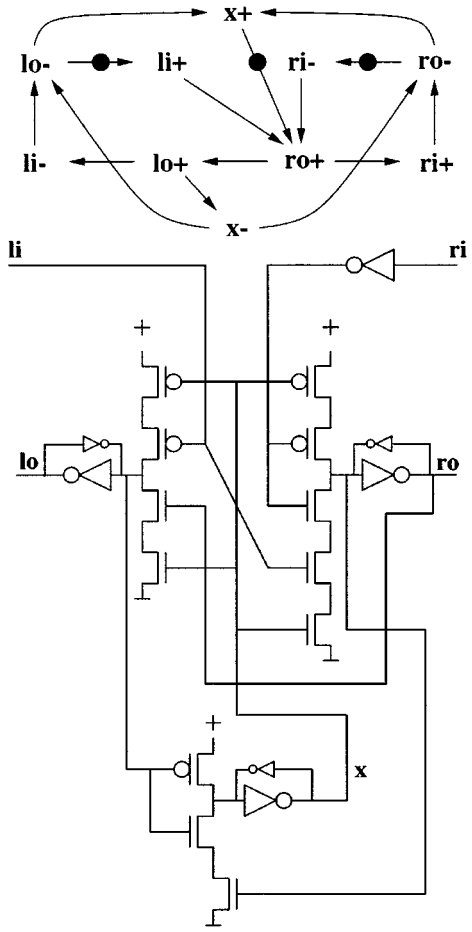


Fig. 14. Speed-independent specification and circuit.

than the delay of one gate. The most stringent remaining constraint is $x- < r_i+$. In the worst case, both r_i+ and $x-$ will be enabled simultaneously by r_o+. In this case, the delay of $x-$ is required to be shorter than the delay of r_i+ (from the environment). Since we assume that the environment is an identical circuit, it corresponds to requiring that the delay of $x-$ to be shorter than that of r_o+, that is easy to satisfy. In case of a very fast environment, this constraint can still be satisfied by transistor sizing or delay padding for gate x.

The second FIFO (second row) is a speed-independent circuit derived by petrify with *automatic concurrency reduction* [45], and without constraining the concurrency of the input and output signals of the cell in order to preserve the performance as much as possible. The result is shown in Fig. 14, where CSC was obtained through state variable insertion and concurrency reduction. In comparison with the RT circuit, notice the gC elements with two p-transistors in series and the ordering between ro+ and lo+. Because of concurrency reduction only one state signal is required, like in the case of the automatic RT solution. However, the state signal is on the critical cycle and the implementations of lo and ro contain additional p-transistors, which make performance of the speed-independent circuit approximately 18% worse than the RT one. Note that, without concurrency reduction, three state signals would be required to solve all state encoding conflicts and a much larger and slower circuit would result.

TABLE V
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

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

The third and the fourth rows of Table IV report results for relative timing and speed-independent circuits, further optimized for performance by applying De Morgan's laws. It can be observed again that the optimized RT circuit is approximately 25% faster than the optimized speed-independent design.

C. RAPPID Control Circuits

This section compares manually optimized RT control circuits used for RAPPID [22], [12] with those automatically derived by petrify. For each example, Table V reports: manual (obtained by applying relative timing manually), automatic (obtained automatically by petrify and applying relative timing), and speed-independent (obtained automatically by petrify without concurrency reduction).

Results in the table show 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 III. 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. Moreover, the optimization goal for these circuits was *performance*, and hence we claim that the automated implementation was *not worse than the manual design in any case.*

D. Impact of Early Enabling Assumptions

The same experiments presented in Table III have been run by not using early enabling assumptions. The overall results in circuit complexity (total number of literals) are the following:

- specifications without CSC (Table III(a)): 330 literals;
- specifications with CSC (Table III(b)): 248 literals.

Thus, early enabling assumptions still contribute to improve the quality of the circuits in about 10% for those specifications with CSC. This improvement also affects the speed of the circuit.

For those specifications without CSC, the impact is very modest. This is mainly due to the fact that petrify does a good job in inserting new state signals by trying to increase their concurrency. This gives less margin to take advantage of the potential concurrency of early enabling assumptions.

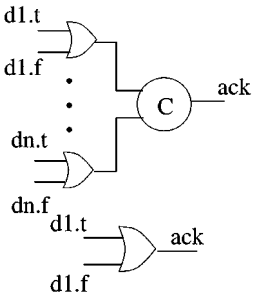


Fig. 15. Optimization of a completion detector using simultaneity assumptions.

E. Impact of Simultaneity Assumptions

The same experiments have also been run by not using simultaneity assumptions. The results have shown that the impact of these assumptions is negligible for the benchmarks in Table III. This is mainly due to the fact that most benchmarks in the table are of reactive nature, where inputs mostly trigger outputs, and outputs mostly trigger inputs. Hence direct causal relations between outputs (a necessary condition for applying simultaneity assumptions) are infrequent. In other words the considered control circuits are quite shallow, and this constrains the applicability of optimization based on simultaneity. Nevertheless, we do believe that the notion of simultaneity is important for optimization, as shown by the example in Fig. 5. A similar situation occurs with other benchmarks, such as hazard, when simultaneity assumptions are applied to input events.[11] It allows the designer to change the dependencies between causally unrelated events. This is a way to formally justify delay matching, a technique that is often used for design of asynchronous data paths, as shown in Fig. 15. The same result of optimization can be obtained formally by applying simultaneity assumption to all data bits with respect to the completion detector signal. Although design of data paths is not the main topic of this paper, such capability indicates potential power of the simultaneity assumption for larger control circuits and especially *control circuits with symmetries*.

IX. Conclusion

Lazy transition systems have been proposed as a computational model for timed circuit synthesis, where the notions of enabling and firing are distinguished for a signal switching event. In this design flow, necessary synthesis conditions, a synthesis algorithm, and a method to derive a sufficient set of timing constraints for correctness have also been proposed.

The main results of this work can be summarized as follows.

- Two types of relative timing assumptions, difference (one-sided) and simultaneity (two-sided), are used.
- Timing information is defined in terms of relations among events rather than absolute delays of individual events. In this way, reasoning about the observable behavior of the system is much more efficient.
- The don't care space used for optimization is determined either by unreachability, i.e., reduction of the state space, or by laziness, i.e., expansion of the enabling region.
- The method allows the timing assumptions to be either provided by the designer or derived automatically by synthesis or analysis tools. The second feature is especially interesting for its applicability to those events that are not observable in the original specification, e.g., events of internal signals used for state encoding or logic decomposition.
- Satisfaction and verification of timing constraints (i.e., timing assumptions actually used by optimization) is left to the designer's responsibility. Some existing tools can assist in solving such task [46], [20].

This approach helps bridging two critical gaps in the synthesis of control circuits. The first gap is between the two main approaches for automated asynchronous controller synthesis, those based on fundamental mode (global timing constraints) and those based on IO mode. It also allows asynchronous circuits to exploit available timing information, rather than always making worst case assumptions about the relative delays of gates (e.g., assuming that one gate may be slower than a sequence of three gates may be excessive in several technologies). Moreover, the exploitation of the idea of early enabling allows the synthesis process to maximize performance by increasing the effective amount of concurrency in the system.

[11]Conservatively, petrify never assumes simultaneity for input events. These assumptions must be provided by the designer when it is known that the environment behaves according to the assumption.

Acknowledgment

The authors would like to thank S. Rotem for initiating this research.

References

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

[2] J. M. Chris, "Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits ," Ph.D. dissertation, Dept. Elec. Eng., Stanford Univ., 1995.

[3] M. N. Steven, "Automatic Synthesis of Burst-Mode Asynchronous Controllers ," Ph.D. dissertation, Stanford Univ., Department of Computer Science, 1993.

[4] Y.-C. Chantal, L. Bill, and M.d. Hugo, "ASSASSIN: A synthesis system for asynchronous control circuits,", Tech. Rep., IMEC, 1994.

[5] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Trans. Information Syst.*, vol. E80-D, pp. 315–325, Mar. 1997.

[6] D. A. Huffman, "The synthesis of sequential switching circuits," *J. Franklin Inst.*, vol. 257, pp. 161–190, Mar. 1954.

[7] S. H. Unger, *Asynchronous Sequential Switching Circuits*. New York: Wiley, 1969.

[8] E. M. David and W. S. Bartky, "A theory of asynchronous circuits," in *Proc. Int. Symp. Theory of Switching*. Cambridge, MA: Harvard Univ. Press, 1959, pp. 204–243.

[9] B. Coates, A. Davis, and K. Stevens, "The post office experience: Designing a large asynchronous chip," *Integration, VLSI J.*, vol. 15, no. 3, pp. 341–366, Oct. 1993.

[10] K. Y. Yun, "Synthesis of Asynchronous Controllers for Heterogeneous Systems ," Ph.D. dissertation, Stanford Univ., 1994.

[11] C. W. Moon, P. R. Stephan, and R. K. Brayton, "Synthesis of hazard-free asynchronous circuits from graphical specifications," in *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 1991, pp. 322–325.

[12] K. S. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. J. Myers, K. Y. Yun, R. Kol, C. Dike, and M. Roncken, "An asynchronous instruction length decoder," *IEEE J. Solid-State Circuits*, vol. 36, pp. 217–228, Feb. 2001.
[13] T. Henzinger, Z. Manna, and A. Pnueli, "Timed transition systems," in *Proc. REX Workshop Real-Time: Theory in Practice*, vol. 600, LNCS. New York, 1992, pp. 226–251.
[14] C. J. Myers and T. H.-Y. Meng, "Synthesis of timed asynchronous circuits," *IEEE Trans. VLSI Syst.*, vol. 1, pp. 106–119, June 1993.
[15] R. Alur, "Timed automata," in *NATO-ASI 1998 Summer School Verification of Digital and Hybrid Syst.*, 1998.
[16] D. Sager, M. Hinton, M. Upton, T. Chappell, T. Fletcher, S. Samaan, and R. Murray, "A 0.18 μm CMOS IA32 microprocessor with a 4 GHz integer execution unit," in *ISSCC'2001*: IEEE Press, 2001, pp. 324–325.
[17] H. Henrik and M. B. Steven, "Bounded delay timing analysis of a class of CSP programs with choice," in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, Nov. 1994, pp. 2–11.
[18] M. Bozga, O. Maler, and S. Tripakis, "Efficient verification of timed automata using dense and discrete time semantics," in *CHARME'99*, vol. 1703, Lecture Notes in Computer Science, L. Pierre and T. Kropf, Eds., 1999, pp. 125–141.
[19] N. Radu and P. Ad, "Verification of speed-dependences in single-rail handshake circuits," in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits Syst.*, 1998, pp. 159–170.
[20] M. A. Peña, J. Cortadella, A. Kondratyev, and E. Pastor, "Formal verification of safety properties in timed circuits," in *Proc. Int. Symp. Advanced Res. Asynchronous Circuits and Syst.*, Apr 2000.
[21] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Taubin, and A. Yakovlev, "Lazy transition systems: Application to timing optimization of asynchronous circuits," in *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 1998, pp. 324–331.
[22] S. Ken, G. Ran, and R. Shai, "Relative timing," in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits Syst.*, Apr. 1999, pp. 208–218.
[23] W. Coates, J. Lexau, I. Jones, I. Sutherland, and S. Fairbanks, "Fleetzero: An asynchronous switching experiment," in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits Syst.*. New York: IEEE Computer Soc. Press, 2001.
[24] S. Ivan and F. Scott, "Gasp: A minimal fifo control," in *Proc. Int. Symp. Advanced Res. Asynchronous Circuits Syst.*, Mar. 2001.
[25] C. Jordi, K. Michael, M. B. Steven, and K. Stevens, "Synthesis of asynchronous control circuits with automatically generated timing assumptions," in *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 1999, pp. 324–331.
[26] T. Murata, "Petri Nets: Properties, analysis and applications," *Proc. IEEE*, pp. 541–580, Apr. 1989.
[27] A. Arnold, *Finite Transition Systems*. Englewood Cliffs, NJ: Prentice Hall, 1994.
[28] T.-A Chu, C. K. C. Leung, and T. S. Wanuga, "A design methodology for concurrent VLSI systems," in *Proc. Int. Conf. Computer Design (ICCD)*, 1985, pp. 407–410.
[29] L. Y. Rosenblum and A. V. Yakovlev, "Signal graphs: from self-timed to timed ones," in *Proc. Int. Workshop Timed Petri Nets*, Torino, Italy, July 1985, pp. 199–207.
[30] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky, *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. London, U.K.: Wiley, 1993.
[31] S. M. Burns, "General condition for the decomposition of state holding elements," in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits Syst.*, Mar 1996.
[32] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, and A. Yakovlev, "Decomposition and technology mapping of speed-independent circuits using Boolean relations," *IEEE Trans. Computer-Aided Design*, vol. 18, Sept. 1999.
[33] T. Nanya, A. Takamura, M. Kuwako, M. Imai, M. Ozawa, M. Ozcan, R. Morizawa, and H. Nakamura, "Scalable-delay-insensitive design: A high-performance approach to dependable asynchronous systems," in *Proc. Int. Symp. Future Intellectual Integrated Electron.*, Mar. 1999.
[34] P. Vanbekbergen, G. Goossens, and B. Lin, "Modeling and synthesis of timed asynchronous circuits," in *Proc. European Design Automation Conf. (EURO-DAC)*, Sept. 1994, pp. 460–465.
[35] K. McMillan, "Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits," in *Proc. Int. Workshop Computer Aided Verification*, vol. 663 of Lecture Notes in Computer Science, G. v. Bochman and D. K. Probst, Eds., 1992, pp. 164–177.
[36] S. M. Burns, "Performance Analysis and Optimization of Asynchronous Circuits," Ph.D. dissertation, California Inst. Technol., 1991.
[37] K. McMillan and D. Dill, "Algorithms for interface timing verification," in *Proc. Int. Conf. Computer Design (ICCD)*, Oct. 1992.
[38] L. Luciano and S.-V. Alberto, *Algorithms for Synthesis and Testing of Asynchronous Circuits*: Kluwer, 1993.
[39] K. Michael and S. Jørgen, "Characterizing speed-independence of high-level designs," in *Proc. Int. Symp. Advanced Res. Asynchronous Circuits Syst.*, Nov. 1994, pp. 44–53.
[40] A. Kondratyev, J. Cortadella, M. Kishinevsky, E. Pastor, O. Roig, and A. Yakovlev, "Checking signal transition graph implementability by symbolic BDD traversal," in *Proc. European Design and Test Conf.*, Paris, France, Mar. 1995, pp. 325–332.
[41] C. Jordi, K. Michael, K. Alex, L. Luciano, and Y. Alexandre, "A region-baded theory for state assignment in speed-independent circuits," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 793–812, Aug. 1997.
[42] P. A. Beerel, C. J. Myers, and T. H.-Y Meng, "Covering conditions and algorithms for the synthesis of speed-independent circuits," *IEEE Trans. Computer-Aided Design*, Mar. 1998.
[43] K. Alex, K. Michael, and Y. Alex, "Hazard-free implementation of speed-independent circuits," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 749–771, Sept. 1998.
[44] L. Bill, Y.-C. Chantal, and V. Peter, "A general state graph transformation framework for asynchronous synthesis," in *Proc. European Design Automation Conf. (EURO-DAC)* ., Sept. 1994, pp. 448–453.
[45] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, "Automatic synthesis and optimization of partially specified asynchronous systems," in *Proc. ACM/IEEE Design Automation Conf.*, June 1999, pp. 110–115.
[46] S. Chakraborty, D. L. Dill, and K. Y. Yun, "Min-max timing analysis and an application to asynchronous circuits," *Proc. IEEE*, vol. 87, pp. 332–346, Feb. 1999.

Jordi Cortadella (M'88) received the M.S. and Ph.D. degrees in computer science from the Universitat Politècnica de Catalunya, Barcelona, Spain, in 1985 and 1987, respectively.

He is a Professor in the Department of Software of the same university. In 1988, he was a Visiting Scholar at the University of California, Berkeley. His research interests include computer-aided design of VLSI systems with special emphasis on synthesis and verification of asynchronous circuits, concurrent systems and co-design. He has coauthored over 100 research papers in technical journals and conferences.

Dr. Cortadella has served on the technical committees of several international conferences in the field of Design Automation and Concurrent Systems. He organized the 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems as a Symposium Co-Chair.

Michael Kishinevsky (M'95–SM'96) received the M.Sc. and Ph.D. degrees in computer science from the Electrotechnical University, St. Petersburg, Russia.

He was a Research Fellow at the St. Petersburg Mathematical Economics Institute Computer Department, Russian Academy of Science, 1979–1982 and 1987–1989. From 1982 to 1987, he was with a software company. From 1988 to 1992, he was a Senior Researcher at the R&D Coop TRASSA. From 1992 to 1994, he was a visiting Associate Professor at the Department of Computer Science, Technical University of Denmark. From 1994 to 1998, he was a Professor at the University of Aizu, Japan. Since 1998 he has been with the Strategic CAD Labs, Intel Corporation, Hillsboro, OR. His research interests include high-level and asynchronous design, reactive systems, and theory of concurrency. He coauthored two books in asynchronous design and has published over 60 journal and conference papers.

Dr. Kishinevsky has served on the technical program committee at several conferences and workshops.

Steven M. Burns received a B.A. degree in mathematics from Pomona College, in 1984, and M.S. and Ph.D. degrees in computer science from the California Institute of Technology in 1987 and 1991, respectively.

He is a Principal Engineer at Intel Corporation's Strategic CAD Labs. Prior to joining Intel in 1996, he was an Assistant Professor of the Department of Computer Science and Engineering at the University of Washington, joining the faculty in 1991. His research and development interests include timing analysis and optimization of high-performance digital circuits.

In 1992, Dr. Burns received an NSF Young Investigator Award. He received a best paper award at the 1993 IEEE International Conference on Computer Design and at the 1996 Internation Symposium on Advanced Research in Asynchronous Circuits and Systems. He has served on the technical program committee at several conferences and workshops.

Alex Kondratyev (M'94–SM'97) received the M.S and Ph.D. degrees in computer science from the Electrotechnical University of St. Petersburg, Russia, in 1983 and 1987, respectively.

In 1988, he joined the R&D Coop TRASSA, St. Petersburg, Russia, where he was a Senior Researcher. From 1993 to 1999, he was an Associate professor of the Hardware Department at the University of Aizu. In 2000, he joined Theseus Logic as a Senior Scientist. Currently he is a Research Scientist at Cadence Berkeley Laboratories. He has coauthored a book on formal methods for asynchronous design and has published over 50 journal and conference papers. His research interests include formal methods in system design, synthesis of asynchronous circuits, computer-aided design methodology and theory of concurrency.

Dr. Kondratyev was a co-chair of Async'96 Symposium, co-chair of CSD'98 Conference, and has served as a member of the program committee for several conferences.

Luciano Lavagno (M'93) graduated magna cum laude in electrical engineering from Politecnico di Torino, Italy, in 1983. In 1992, he received the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley.

From 1984 to 1988, he was with CSELT Laboratories, Torino, Italy. In 1988, he joined the Department of Electrical Engineering and Computer Science of the University of California at Berkeley, where he worked on logic synthesis and testing of synchronous and asynchronous circuits. He is the author of a book on asynchronous circuit design, the co-author of a book on hardware/software co-design of embedded systems, and has published over 60 journal and conference papers. Between 1993 and 1998 he was an Assistant Professor with the Department of Electronics of Politecnico di Torino. Since 1993, he has been the architect of the POLIS project, developing a complete hardare/software co-design environment for control-dominated embedded systems. He is currently an Assistant Professor at the University of Udine, Italy and a Research Scientist at Cadence Berkeley Laboratories. He has also been a consultant for various EDA companies, such as Synopsys and Cadence. His research interests include the synthesis of asynchronous and low-power circuits, the concurrent design of mixed hardware and software systems, and the formal verification of digital systems.

In 1991, Dr. Lavagno received the Best Paper award at the 28th Design Automation Conference in San Francisco, CA. He has served on the technical committees of several international conferences in his field (namely the Design Automation Conference, the International Conference on Computer Aided Design, and the European Design Automation Conference).

Kenneth S. Stevens (SM'99) received the B.A. degree in biology in 1982 and the B.S. and M.S. degrees in computer science, in 1982 and 1984, from the University of Utah. He received the Ph.D. degree in computer science from the University of Calgary, AB, Canada, in 1994.

From 1984 through 1991, he held research positions at the Fairchild/Schlumberger Laboratory for AI Research, the Schlumberger Palo Alto Research Laboratory, and Hewlett Packard Laboratories, in Palo Alto, CA. He became an Assistant Professor at the Air Force Institute of Technology in Dayton, OH in 1994, and since 1996 he has been an Adjunct Professor. Since 1996 he has been employed at Intel Corporation's Strategic CAD Labs in Hillsboro, OR, where he is currently a Principal CAD Engineer. His research interests include asynchronous circuits, VLSI, architecture, hardware synthesis and verification, and timing analysis. He holds seven patents and has been the principal author for three papers which received the best paper award.

Dr. Stevens has been on the Technical Program Committee for the Async conference series since 1998.

Alexander Taubin (M'94–SM'96) received the M.Sc. and Ph.D. degrees in computer science and engineering from Electrotechnical University of St.Petersburg, Russia.

He was a Research Fellow at the Computer Department of St. Petersburg Mathematical Economics Institute, USSR Academy of Science, from 1979 to 1989. From 1988 to 1993, he was a Senior Researcher at the R&D Coop. TRASSA. From 1993 to 1999, he was with the Department of Computer Hardware at the University of Aizu Japan, as a Professor. In 1999, he joined Theseus Logic, Inc. Sunnyvale, CA as Senior Scientist. His current research interests include design of asynchronous systems (analysis, synthesis, testing, formal verification and architectural design for asynchronous microprocessors and DSP) and models for concurrent behavior. He coauthored two books in asynchronous design and has published more then 40 journal and conference papers.

Dr. Taubin has served on the technical committees of several international conferences in his field.

Alexandre Yakovlev (S'94–M'97) received the M.Sc. and Ph.D. degrees in computing science from Electrotechnical University of St. Petersburg, Russia, in 1979 and 1982, respectively.

He has worked at Electrotechnical University of St. Petersburg in the area of asynchronous and concurrent systems since 1980, and in the period between 1982 and 1990 held positions of Assistant and Associate Professor in the Computing Science Department. Since 1991, he has been a Lecturer, Reader, and since 2000 Professor in Computer Systems Design at the Newcastle University Department of Computing Science, where he is heading the VLSI Design research group. His current interests and publications are in the field of modeling and design of asynchronous, concurrent, real-time and dependable systems. He has coauthored over 100 research papers in technical journals and conferences.

Dr. Yakovlev has organized and served on the technical committees of several international conferences in the field of asynchronous systems, concurrency and Petri nets.