Bridging modularity and optimality: delay-insensitive interfacing in asynchronous circuits synthesis *

Hiroshi Saito, Univ. of Aizu, Japan
Alex Kondratyev, Univ. of Aizu, Japan
Jordi Cortadella, Univ. Politècnica Catalunya, Spain
Luciano Lavagno, Univ. of Udine, Italy
Alexander Yakovlev, Univ. of Newcastle upon Tyne, UK

Abstract

Two trends are of major concern for digital circuit designers: the relative increase of interconnect delays with respect to gate delays and the demand for design reuse. Both pose difficult problems to synchronous design styles, and can be tackled more naturally within the asynchronous paradigm. Unfortunately even in asynchronous design the normal hypotheses about the delays of gates and wires are often overly optimistic. One of the popular assumptions is to consider gate delays to be arbitrary while neglecting the skew in wire delays (so-called speed-independence (SI) assumption). Taking wire delays into account is possible and in its extreme leads to delay-insensitive (DI) implementations which work correctly under any wire delay distribution. However, such implementations are costly.

This work suggests to separate all on-chip interconnections into two classes: local (for which the delays can be under control) and global (with arbitrary delays). This leads to locally SI globally DI implementations which are more practical than fully DI circuits and are in better correspondence with technology parameters than fully SI circuits. Our approach allows logic synthesis to proceed independently for all the locally SI blocks and yields functionally correct circuits without requiring *any synthesis/layout iteration or interaction*. This simplifies dramatically the *timing convergence problem* for ASICs.

We tackle the problem at the behavior level and develop a simple transformation which ensures delay-insensitive properties for particular wires. The method is illustrated by a realistic design example. The preliminary experimental results show that the area and performance penalty are within 40% and 20% respectively.

1 Introduction

Asynchronous systems, free from the clock, offer a number of potential advantages in Deep-Sub-Micron digital and mixed-signal design. They include robustness of designs to technology variations, greater modularity and capability for component reuse. These factors are essential in complex applications where complete redesign for a localized functionality change becomes unrealistic, and where time-to-market is crucial.

Two subclasses of asynchronous circuits are known to be able to sustain certain parameter variations: *speed-independent (SI) circuits* [7] and *delay-insensitive (DI) circuits* [11]. The former are characterized by the fact that their behaviour is insensitive to gate delays (these can have arbitrary value) but assume wire delays to satisfy the following condition (isochronicity of forks): the max delay of a wire after a fork must be less than the min gate delay. DI circuits allow wire delays to have arbitrary values. Although DI circuits are clearly much more attractive for the Deep Sub-Micron technology, where wire delays are as significant as gate delays, the domain of functionally useful DI circuits is very limited if one considers them at the level of ordinary logic gates. Thus DI circuits are typically constructed out of macro-modules that consist of several gates [10].

It is therefore quite natural to look for a way of exploiting the advantages of both design strategies, namely the optimality of the SI logic synthesis and the design robustness and DI compositionality of the macro-module method. The target of the synthesis process is therefore deemed to be a *globally DI locally SI* circuit. This approach was suggested, without any concrete implementation strategy, in [12].

Our work has some similarities (and in particular a consistent view of technology trends) with the *wire planning* strategy suggested in [8]. In both cases, logic synthesis is preceded by a "delay-aware" step that partitions the system into blocks where wire delays are smaller than gate delays. However, due to the synchronous nature of the underlying implementation, [8] requires a placement and global routing step before synthesis. On the other hand, in our case only the communication protocol between the blocks must be (automatically) modified to satisfy a set of DI axioms. After that, logic synthesis can proceed independently for all the blocks, without requiring *any synthesis/layout iteration or interaction*. This simplifies dramatically the *timing convergence problem* for asynchronous ASICs.

*This work was partially supported by ESPRIT No. 21949 (ACiD-WG), EPSRC GR/L24038, GR/L28098, and CICYT TIC98-0410.

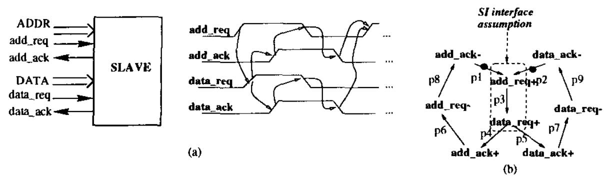

Figure 1: Simple asynchronous interface

The modeling formalism for the suggested design flow is based on Signal Transition Graphs (STG). It is known that from an STG one can derive a speed-independent implementation using different design procedures [1, 4]. In this paper we suggest a behavioral transformation called *order relaxation* which is aimed to allow delay-insensitivity with respect to certain STG events. Based on this transformation an initial specification could be iteratively refined until the desired level of delay-insensitivity is reached.

The rest of the paper is organized as follows. Section 2 contains a theoretical background. The theory behind DI transformation is presented in Section 3. Section 4 shows an application of the suggested methodology to a realistic design example. Section 5 concludes the work.

2 Theoretical background

Figure 1.a shows a simple interface between two modules in an asynchronous system, a master (e.g., a processor) and a slave (e.g., memory). The interface involves two signal handshakes, one for controlling the transmission of an address (add_{req} and add_{ack}) and another for data ($data_{req}$ and $data_{ack}$). The timing diagram shown in Figure 1.a defines the synchronization protocol between the handshakes for the case of writing data into slave. This protocol allows an additional skew compensation between address and data, making sure that the address is delivered to the slave strictly before data, to give an additional delay in the corresponding address decode logic. This condition is captured by an arc directed from the rising edge of the add_{req} signal to that of $data_{req}$.

Figure 1.b shows the Petri Net (PN) corresponding to the timing diagram of the controller. All events in this PN are interpreted as signal transitions: rising transitions of signal a are labeled with "$a+$" and falling transitions with "$a-$". We also use the notation $a*$ if we are not specific about the sign of the transition. Petri Nets with such an interpretation are called *Signal Transition Graphs (or* STG*s)* [1]. STGs are typically represented in a "shorthand" form, where places with one input arc and one output arcs are implicit.

An STG transition is *enabled* if all its input places contain a token. In the initial marking $\{p1, p2\}$ of the STG in Figure 1.c transition $add_{req}+$ is enabled. Every enabled transition can fire, removing one token from every input place of the transition and adding one token to every output place. After the firing of transition $add_{req}+$ the net moves to a new marking, $\{p3\}$, where and $data_{req}+$ becomes enabled.

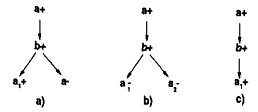

Figure 2: Consistency violations in STG

Transitions in STG could be involved in different ordering relations. We would say that transitions $a*$ and $b*$ are in *direct conflict* if there exists a reachable marking in which both of them are enabled but firing of one of them disables the other. If $a*$ and $b*$ are enabled in some reachable marking but are not in a direct conflict we will call them *concurrent*. Conflict relations can be generalized by considering the transitive successors of directly conflicting transitions. Transitions which are not concurrent and are not in conflict are called *ordered*. An STG is *consistent* if in every transition sequence from the initial marking, rising and falling transitions alternate for each signal.

There are two sources of consistency violation in an STG:

1) *Auto-concurrency*, i.e. concurrency of transitions of the same signal (see Figure 2.a,b) and

2) *Switch-over incorrectness*, which takes place between two ordered rising (falling) transitions which have no falling (rising) transition in between (see Figure 2.c).

The set of all signals STG is partitioned into a set of *inputs*, which come from the environment, and a set of *outputs* and *state* signals that must be implemented.

In addition to consistency, the persistency property is required for an STG to be implementable as a hazard-free asynchronous circuit.

An event a^* is *persistent* in marking m if it is enabled in m and remains enabled in any other marking reachable from m by firing another event b^*. An STG is *output-persistent* if all output signal events are persistent in all reachable markings and input signals cannot be disabled by outputs. Output persistency therefore only allows input events to be in direct conflict.

The following important statement was proved in [1]: *an STG can be implemented by a speed-independent circuit if it is consistent and output-persistent.*

3 Construction of Delay-Insensitive Interface

A conventional definition of delay-insensitivity is based on satisfying the following axioms [11, 5] in a behavioral description:

1. *No auto-concurrency*
2. *Alternating input/output* (input events can only immediately precede output events and output events can only immediately precede input events)

3. *No cross-disabling* (inputs and outputs cannot disable each other)

In this work we will relax the above axioms taking into account specific features of the targeted task:

- The investigation is focused not on *total* delay-insensitivity but on the *delay-insensitive interfacing* only (the basic assumption is that within a module a designer or physical design tool can keep wire delays under control and hence there is no point to ensure delay-insensitivity at the level of events internal to a module).
- Contrary to conventional approaches to DI synthesis the tasks of designing a module and its environment are considered separately. It results in asymmetry of requirements which are imposed on DI interface: only inputs are required to be accepted in a delay-insensitive fashion because delay-insensitivity with respect to outputs matters only when an implementation for the environment is synthesized. Of course, symmetry is re-established if all modules are synthesized in this fashion.

Informally the above conditions are illustrated by Figure 3 where the suggested design approach is targeted at an interface scheme that should be robust to wire delay variations.

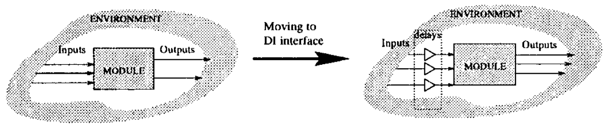

Figure 3: Delay-insensitive interfacing

Based on that scheme we define *delay-insensitive interfacing.*

Definition 3.1 *A specification of a module satisfies delay-insensitive interfacing if it meets the following conditions:*

1. *No auto-concurrency*
2. *Alternating inputs (input events can only immediately precede outputs events)*
3. *No cross-disabling*

Our design framework uses STGs as a model basis. The natural question is: what are the implications of the requirements of delay-insensitive interfacing for the properties of the original STG?

Proposition 3.1 *A consistent and speed-independent STG satisfies DI interfacing conditions if no input transition directly precedes another input transition.*

The proof is trivial: non-auto-concurrency is a necessary condition of STG consistency, absence of cross-disabling is guaranteed by speed-independence and alternation of inputs directly comes from the conditions of the proposition.

Proposition 3.1 gives an idea of where DI interfacing may be violated in an STG: these are STG fragments in which input transitions are directly causally related. After adding arbitrary delays into every input wire (see Figure 3) a given module may receive originally sequenced inputs in any order. The latter means that from the module point of view such inputs are concurrent. Hence a possible transformation strategy for an STG towards DI interfacing removes direct causal dependencies between inputs and making them concurrent. This transformation might be performed by iterative application of a simple operation which we will call *order relaxation* and illustrate by Figure 4. Informally order relaxation removes a causal arc between events a and b making them concurrent while keeping other ordering relations as much as possible.

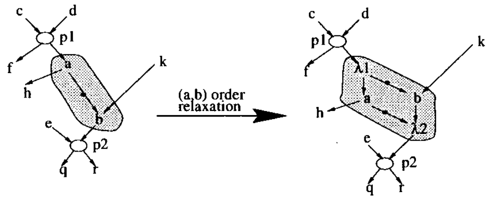

Figure 4: Order relaxation

To investigate the nature of order relaxation let us return to the role of ordering relations between STG events.

STG usually specifies a cyclic process in which the same STG events might fire many times (could have many instances). An STG behavior is formally specified by a set of feasible sequences (traces) in which different instances of the same event are distinguished by their index ($a^1, a^2, \ldots$ e.g.). When considering ordering relations one can divide the set of traces into equivalent classes. Traces from the same equivalent class has the same set of event instances and differ only in some interleavings of concurrent transitions. Such equivalent classes are called concurrent runs [6]. It is convenient to introduce ordering relations at the level of event instances and concurrent runs: we will say that a^i *precedes* b^j in concurrent run ξ if in any trace corresponding to run ξ, a^i occur before b^j.

The following two properties of order relaxation help to understand better the transformation towards DI interfacing.

Property 3.1 *Order relaxation between events a and b preserves pairwise ordering relations in concurrent runs between all instances except for instances of a and b.*

Proof: Let us consider the case when $a \rightarrow b$ (like in Figure 4). Suppose that the statement of the property is wrong. Then one could find a pair of instances c^i and d^j such that c^i precedes d^j in a concurrent run ξ of original STG D but not in run ξ' of STG D', obtained through the order relaxation by (a, b) (ξ and

ξ' have the same set of event instances). The latter means that in D' there exists a trace σ in which d^j occurs before c^i.[1] By reordering concurrent events in σ let us obtain a trace $\sigma' = \gamma, \delta, d^j, \eta, c^i$ such that γ is a feasible signal sequence in the original D and it has a maximal length among all possible permutations within σ.

The transformation coming from the order relaxation is localized in the vicinity of events a and b. Signal sequence γ is feasible in both D and D'. From Figure 4.a one can see that the only event that is enabled after γ in D' but is not enabled in D is the event b. Therefore subsequence δ should start from some instance b^k of b. Whenever an instance b^k becomes enabled the corresponding instance a^m of a also becomes enabled ($m = k$ if the arc $(\lambda 1, b)$ is not initially marked and $m = k - 1$ if the arc $(\lambda 1, b)$ is initially marked).

Case 1. Let a^m belong to $\delta \cup \eta$. Then it can be moved in the trace to the position immediately before b^k, which results in a sequence γ, a^m, b^k that is feasible in both D and D'. This sequence is larger than γ which contradicts the choice of γ.

Case 2. Let $a^m \notin \delta \cup \eta$. Then the corresponding instance of dummy event $\lambda 2$ is not enabled and as $\lambda 2$ is the only successor of b then no events in STG D' could notice the results of early firing of b^k with respect to D. Therefore, none of them could change the ordering. □

Property 3.2 *Order relaxation between two events preserves output persistency in STG.*

Proof: Let D' be an STG obtained after order relaxation by events a and b e.g. If the statement of Property was wrong one could find a marking M reachable in D' in which two events c and d are enabled and firing of one of them disables the other leading to the output persistency violation. Moreover this pair of events should not be in conflict in the original STG D because D is output persistent. Clearly the disabling between c and d is possible only if they are sharing some input place p' in D'.

Case 1. $\{c, d\} \cap \{a, b\} = \emptyset$. Then place p' is outside the scope of this order relaxation transformation and therefore is in one-to-one correspondence with some place p in D. This place in D is shared by c and d but no disabling happens in their firing and hence c and d are never concurrent in their consumption of p (they should consume a tokens from p in turn, i.e. orderly). As ordering relations are preserved in order relaxation then c and d in D' also should not be concurrently enabled, which contradicts the assumption about their conflict.

Case 2. If c or d coincides with a or b the new (in comparison to D) conflicts never arise because the order relaxation is conflict non-increasing by a and b (see Figure 4). □

[1]In the following we will consider only signal traces, i.e. the traces in which all dummy events are deleted.

When in the original STG two inputs are directly causally related, then according to Definition 3.1 DI interfacing can only be obtained by applying order relaxation to them. The latter by Property 3.2 does not cause any new cross-disablings to occur. Unfortunately not all the requirements of DI interfacing are safely preserved during order relaxation. Indeed if events a and b correspond to transitions of the same signal their order relaxation immediately produces auto-concurrency. In case this does not happen the above transformation is *strictly delay-insensitivity increasing* and by iterative application of it eventually (if non-auto-concurrency is preserved) all the requirements of DI interfacing should be met in the specification.

Example 3.1 *Let us illustrate the transformations to ensure the DI interfacing using the example of asynchronous controller from Section 2. The original specification is shown in Figure 5.a. For this specification DI interfacing is violated by a direct causal dependencies between input transitions $add_{req}+$ and $data_{req}+$ (this violations is denoted in Figure 5.a by shading). The violations can be removed by performing order relaxation between input events.*

The order relaxation between $add_{req}+$ and $data_{req}+$ results in the removal of the arc $(add_{req}+, data_{req}+)$, adding direct predecessors of $add_{req}+$ to $data_{req}+$ (i.e. $data_{ack}- \rightarrow data_{req}+$, $add_{ack}- \rightarrow data_{req}+$) and adding direct successors of $data_{req}+$ to $add_{req}+$ (i.e. $add_{req}+ \rightarrow data_{ack}+$ and $add_{req}+ \rightarrow add_{ack}+$) [2]. *The obtained specification is non-auto-concurrent and satisfies the requirements of DI interfacing. By STGs Figure 5.a.b one can obtain corresponding SI and DI implementations which are shown in Figure 5.c.d. The complexity of logic for these two implementations is: SI - 8 literals, DI - 11 literals. This means a 37% of area penalty for DI interfacing.*

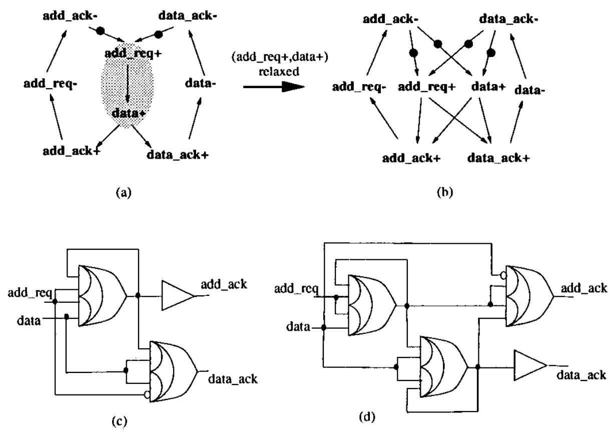

Figure 5: Illustration of SI to DI transformation

[2]Wherever it simplifies the result of transformation we would drop the dummy transitions $\lambda 1$ and $\lambda 2$ from Figure 4.

4 Case study. Controller for analog-to-digital converter

In this section we present an experiment that has been carried out to test the proposed method and evaluate the cost of DI interfacing in a more practical design example than those considered above.

The example originates from a practical case study in which an asynchronous analog-to-digital converter (ADC) has been developed with a speed-independent controller [3].

This ADC implements a well-known successive approximation algorithm. According to this algorithm, a comparator is iteratively activated to compare the value of the given input voltage with the approximate voltage produced by a digital-to-analog converter (DAC), whose digital input comes from a register, in which the n-bit value is refined bitwise, starting from the most significant bit. Each refining bit is produced by a one-bit buffer connected to the output of the comparator. The use of asynchronous logic allows this system to avoid synchronization errors due to meta-stability (which is known to be a problem in clocked converters), which may arise in the analog part of the circuit, and to smooth out the temporal effect of potential meta-stability resolution [3] one the whole period of conversion.

The central part of the asynchronous ADC, which controls copying a bit value from the one-bit buffer to the n-bit register with a single bit shift, is an n-way scheduler; it is functionally similar to a classical pulse distributor. The scheduler's behaviour can be specified by an STG whose structure is regular. The specification of a scheduler with 3 cells is shown in Figure 6.a.

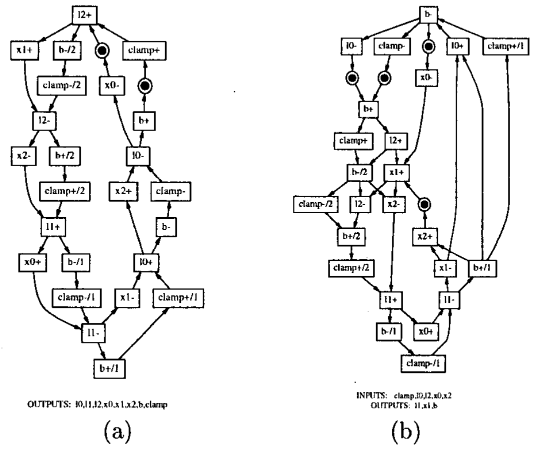

Figure 6: A specification of 3-cell scheduler (a) and the input order relaxation for the cell 1 (b)

From the analysis of the causal relations between events one could see that the behavior of the i-th cell of the scheduler depends on the state of the (i-1)-th and (i+1)-th cells together with the signal *clamp* (output of some completion detection logic in a storage buffer; see Figure 7.a). Hence the speed-independent implementation of the scheduler might be obtained directly using the STG of Figure 6, which gives the following logic circuit:

$l_i = clamp\ (\overline{x}_{i-1}\overline{x}_{i+1} + l_i) + l_i\overline{x}_{i-1}$;
$x_i = \overline{x}_{i-1}x_i + l_i + l_{i+1}$;
$b = \overline{l}_1\overline{l}_2\ldots\overline{l}_n$

The drawback of the SI implementation is that the designer is responsible for satisfying the SI assumptions about wiring delays between scheduler cells.

In case of conversion with a data path (with many cells in the scheduler) or in order to increase the flexibility of layout, it could be more convenient to partition the whole circuit of the scheduler into smaller parts which could be placed in different positions on the chip (not necessarily adjacent). Then within each part the designer could still rely on the SI hypothesis about the wiring between cells but in the interface between these parts the wire delays could be large and we need a more conservative approach. In the extreme, interface delays are assumed to be arbitrary which leads to DI interfacing and gives the scheduler structure shown in Figure 7.b.

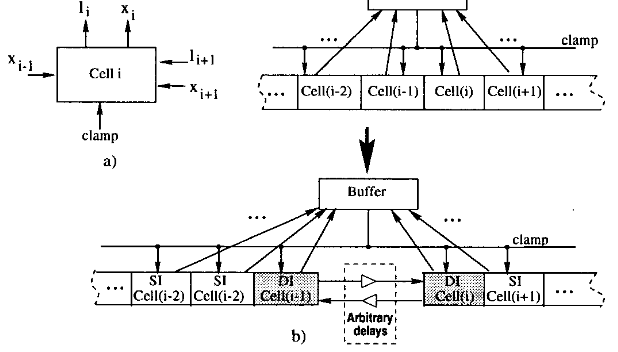

Figure 7: A scheduler circuit structure

In order to evaluate an upper bound for the cost of partitioning the scheduler we consider the smallest possible parts (containing one cell each). Each cell communicates with its neighbors in a DI fashion and therefore synthesis of such a scheduler reduces to the task of DI interfacing between cells. Synthesis could be done via an order relaxation on the STG in Figure 6, where for the i-th cell all the transitions of the (i-1)-th and (i+1)-th cells are relaxed. The result is shown in Figure 6.b. From this STG the following logic equations can be derived:

$l_i = clamp\ (\overline{x}_{i-1}\overline{x}_{i+1}x_ib + l_i) + l_i\overline{x}_{i-1}$;
$x_i = \overline{x}_{i-1}(x_i + x_{i+1}l_{i+1}) + l_i$;
$b = \overline{l}_1\overline{l}_2\ldots\overline{l}_n(b + \overline{clamp}) + b\ \overline{clamp}$

A comparison between the SI and DI implementations shows that the latter is about 38% larger(see

SI implementation		DI implementation	
area (lits)	performance (ns)	area (lits)	performance (ns)
34	10709	47	11422

Table 1: Comparison of SI and DI implementations of the scheduler

Table1).

We have also analyzed performance for the SI and DI implementations, using *logic simulation*. We have synthesized both the scheduler circuit and its environment and simulated the resulting autonomous system. From Table1 one could see that the degradation of performance because of the increased complexity of DI implementation is about 7%.

It is worth noting that these number are significantly lower than those usually reported when referring to synthesis results for DI implementations. The reason for that lies in our more flexible design strategy, that is *speed-independent circuits with DI interfacing instead of totally DI solutions.*

5 Conclusions

Design styles which neglect wire delays seem to be overly optimistic even with current technology and will most likely become less and less applicable when moving to deep sub-micron implementations. The extreme case when wire delays are assumed to have arbitrary values leads to the well known delay-insensitive approach for circuit design. However delay-insensitive circuits are often unusable because of their excessive area and performance overheads. In this paper we suggested an approach which results in *partial* delay-insensitivity of an implementation. Under this approach the designer or floor-planning tool identifies a set of *long wires*, which should be implemented in delay-insensitive fashion while for the rest of a circuit other (more conventional) design styles might be applied. In particular, we used speed-independent implementation for the parts of a system in which wire delays could be controlled by the designer or a routing tool, and then applied the delay-insensitive hypothesis only to the wires running between such speed-independent "islands" [8]. These wires then can be routed to any distance, *without affecting the functionality of the circuit* (only, of course, its performance), thus dramatically speeding up timing convergence for asynchronous ASICs.

We have developed an automatic method which ensures the DI requirements by using behavior transformations. To the best of our knowledge, this is the first method which produces a *Delay Insensitive implementation* from a formal specification by using a *highly optimizing synthesis-based procedure*. We believe that this could give a significant reduction in area and performance penalties in comparison to the conventional DI methods which are based on *direct translation* of the initial description into the circuit by using pre-defined library modules, followed at most by a conservative peephole optimization.

It is possible to extend this approach to a more aggressive optimization (for both area and speed) strategy than the SI one, to partially compensate the costs of the DI interfacing. It is based on the use of relative timing at the module level, which can be gradually introduced into the SI logic [2, 9].

REFERENCES

[1] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications.* PhD thesis, MIT, June 1987.

[2] 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. International Conf. Computer-Aided Design (ICCAD)*, pages 324–331, November 1998.

[3] D. J. Kinniment, B. Gao, A. V. Yakovlev, and F. Xia. Toward asynchronous A-D conversion. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 206–215, 1998.

[4] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* Series in Parallel Computing. John Wiley & Sons, 1994.

[5] S. C. Leung and Hon F. Li. A syntax-directed translation for the synthesis of delay-insensitive circuits. *IEEE Transactions on VLSI Systems*, 2(2):196–210, June 1994.

[6] A. Mazurkiewicz. Concurrency, modularity and synchronization. In *Lecture Notes in Computer Science, Vol. 379.* Springer-Verlag, 1989.

[7] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[8] R.H.J.M. Otten and R.K. Brayton. Planning for performance. In *Proceedings of the Design Automation Conference*, June 1998.

[9] K. S. Stevens, S. Rotem, S. M. Burns, J. Cortadella, R. Ginosar, M. Kishinevsky, and M. Roncken. Cad directions for high performance asynchronous circuits. In *Proc. ACM/IEEE Design Automation Conference*, June 1999. (Invited paper).

[10] I. E. Sutherland. Micropipelines. *Communications of the ACM*, June 1989. Turing Award Lecture.

[11] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.

[12] A. V. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.