PAPER Special Section on VLSI Design and CAD Algorithms

Design of Asynchronous Controllers with Delay Insensitive Interface

Hiroshi SAITO[†a)], ***Student Member***, **Alex KONDRATYEV**[††], **Jordi CORTADELLA**[†††], **Luciano LAVAGNO**[††††], **Alex YAKOVLEV**[†††††], ***Nonmembers***, ***and*** **Takashi NANYA**[†], ***Regular Member***

SUMMARY Deep submicron technology calls for new design techniques, in which wire and gate delays are accounted to have equal or nearly equal effect on circuit behavior. Asynchronous speed-independent (SI) circuits, whose behavior is only robust to gate delay variations, may be too optimistic. On the other hand, building circuits totally delay-insensitive (DI), for both gates and wires, is impractical because of the lack of effective synthesis methods. The paper presents a new approach for synthesis of *globally* DI *and locally* SI circuits. The method, working in two possible design scenarios, either starts from a behavioral specification called Signal Transition Graph (STG) or from the SI implementation of the STG specification. The method locally modifies the initial model in such a way that the resultant behavior of the system does not depend on delays in the input wires. This guarantees delay-insensitivity of the system-environment interface. The suggested approach was successfully tested on a set of benchmarks. Experimental results show that DI interfacing is realized with a relatively moderate cost in area and speed (costs about 40% area penalty and 20% speed penalty).

key words: *DI interface, signal transition graph, hazards, behavioral and gate-level transformations*

1. Introduction

As the scale of integration increases, managing synchronization and control of computation and communication on deep sub-micron (DSM) integrated circuits using a global clock is becoming increasingly difficult. Asynchronous systems, free from the clock, offer a number of potential advantages, such as reduced risk of synchronization failures, low power consumption, improved noise and electro-magnetic compatibility to name but a few.

Interpreted Petri Nets called Signal Transition Graphs (STGs) [1] are widely used in specifying an asynchronous system behavior. It is known [1] that from an STG one can derive an implementation that behaves correctly under any distribution of gate delays, i.e. it is speed-independent (SI). The main drawback of SI circuits is in neglecting the influence of wire delays on circuit behavior. For the DSM technology, where wire and gate delays can become equally important, the implementation should be targeted at delay-insensitive (DI) circuits [2], which allow wire delays to be of arbitrary value. In fact, a reasonable strategy for future technologies would require one to partition the system into blocks of relatively small size, for which the designer can keep control of wire delays (SI blocks e.g.) [3], [4], with a DI interface between blocks [5]. Such an approach does not restrict the designer in choosing a particular way of implementing blocks. It works equally well under implementations that are not necessarily SI. For example, it is consistent with the recent development of the globally asynchronous locally synchronous (GALS) design paradigm [6]. However because of the well-developed CAD support for speed-independent circuits [7] this paper mainly targets SI implementations with DI interface.

Two approaches for synthesis are explored. The first one develops a set of behavioral transformations for refining an STG to satisfy DI interfacing requirements (see Fig. 1(a)). A circuit with DI interface might be obtained from the refined STG via the known synthesis methods implemented in the tool Petrify [7]. This framework appears to be efficient and shows overheads

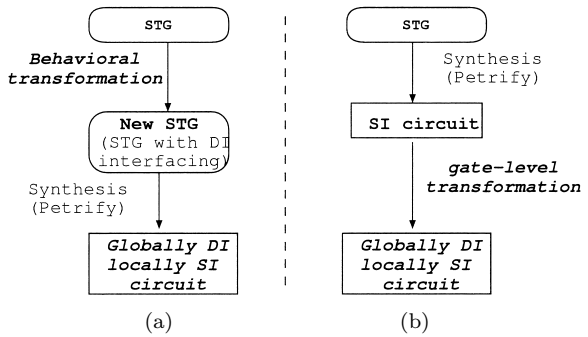

Fig. 1 Design flows for behavioral (a) and gate-level (b) methods.

Manuscript received March 18, 2002.
Manuscript revised May 20, 2002.
Final manuscript received June 13, 2002.

[†]The authors are with the Research Center of Advanced Science and Technology, The University of Tokyo, Tokyo, 153-8904 Japan.
[††]The author is with the Cadence Berkeley Laboratories, Berkeley, CA 94704 U.S.A.
[†††]The author is with the Universitat Politecnica de Catalunya, Barcelona, 08034 Spain.
[††††]The author is with the Politecnico di Torino, Torino, 10129 Italy.
[†††††]The author is with the University of Newcastle upon Type, Newcastle, NE1 7RU U.K.
a) E-mail: hiroshi@hal.rcast.u-tokyo.ac.jp

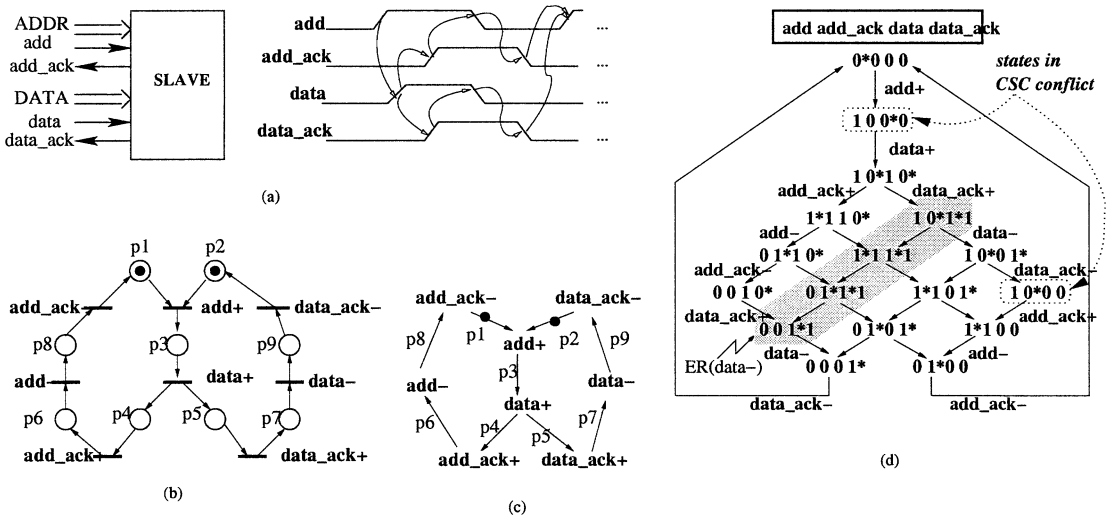

Fig. 2 Simple asynchronous interface: (a) timing diagrams, (b) PN, (c) STG, (d) SG.

of about 40% in area and 20% in speed when ensuring DI interfacing.

The drawback of the behavioral approach is however in its high computational complexity coming from the necessity to explore the reachable state space of the refined STG. Therefore, as an alternative, we also investigate a gate-level approach to designing SI circuits with DI interface. It uses a speed-independent implementation as a starting point and modifies the implementation locally, only for those gates that might "suffer" from changing the SI assumption about communication delays to the DI assumption (see Fig. 1(b)). The approach is based on heuristic constraining the logic functions of such gates by reducing their ON-sets with additional literals. Though this transformation lacks the global view on the optimization process it often produces results that compare favorably with the behavioral approach.

Pure behavioral and pure gate-level methods are two extremes in tackling the DI interfacing problem. The combination of those suggests a nice trade off between optimality and computational complexity and improves the overall flexibility of the design flow.

This work focuses on the automatic introduction of DI interfaces in the control part of the design. There are several possible approaches to handling the data part as well.

1. The data-path can be designed using a DI-encoding (e.g., dual rail, Sperner codes etc. [8]).
2. If a more area-efficient approach, such as bundled data, is chosen for the data-path, like it was in Micropipelines [9], the ordering conditions between data and a corresponding request signal are simpler to satisfy than the ordering conditions between several control signals, possibly coming from different parts of the overall design.

In the rest of the paper we briefly discuss the theory of behavioral transformations to ensure DI interfacing (Sect. 3) with experimental results (Sect. 4). Then we explore a gate-level approach (Sect. 5) for DI interfacing and provide an experimental comparison between these two (Sect. 6).

2. Theoretical Background

Figure 2(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 and add_{ack}) and another for data ($data$ and $data_{ack}$). The timing diagram shown in Fig. 2(a) defines the synchronization protocol between the handshakes for the case of writing data into the slave.

Figure 2(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 and one output arc 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 Fig. 2(c) transition $add+$ 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 $add+$ the net moves to a new marking, $\{p3\}$, where $data+$ becomes enabled.

Transitions in STG could be involved in different ordering relations. Transitions $a*$ and $b*$ are in *direct*

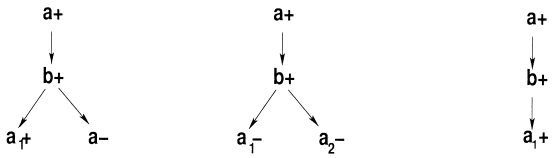

Fig. 3 Consistency violations in STG.

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 direct conflict, they are *concurrent*. Conflict relations can be generalized by considering the transitive successors of directly conflicting transitions. Transitions which are not concurrent and are not in (transitive) conflict are *ordered*.

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.

State graphs. Playing the token game one can generate a *State Graph* (SG) in which each node (a marking) is labeled with a vector of signal values (signals that can change in the state are marked with an asterisk) and arcs between pairs of states are labeled with the corresponding fired transition. A maximally connected set of states in which $a*$ is enabled is called an *excitation region* (ER) for event $a*$ (denoted by ER($a*$), see e.g. the shadowed set of states in Fig. 2(d) corresponding to ER($data-$)). Excitation regions in SG corresponds to transitions in STG. $b*$ is called a *trigger* event wrt $a*$ if in SG by firing $b*$ from some state s outside excitation region ER($a*$ one might reach state s' inside ER($a*$) (enter ER($a*$)). In STG triggering of $a*$ by $b*$ implies the existence of direct causal relations between $a*$ and $b*$, i.e either $b* \rightarrow a*$ or they are mediated by a place.

An SG is *consistent* if in every transition sequence from the initial state, rising and falling transitions alternate for each signal. Figure 2(d) shows the SG for the STG in Fig. 2(c), which is consistent. There are two sources of consistency violation in an STG:

1. *Auto-concurrency*, due to concurrency of transitions of the same signal (see Figs. 3(a), (b)) and
2. *Switchover incorrectness*, due to ordered rising (falling) transitions which have no falling (rising) transition in between (see Fig. 3(c)).

Implementability conditions. In addition to consistency, the following two properties are required for an SG to be implementable as a hazard-free asynchronous circuit. The first property is *speed independence* which reduces to output-persistency of SG events. An event $a*$ is *persistent* in state s if it is enabled in s and remains enabled in any other state reachable from s by firing another event $b*$. An SG is *output-persistent* if all output signal events are persistent in all states and input signals cannot be disabled by outputs.

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.*

The second implementability property, *Complete State Coding* (CSC), is necessary and sufficient for the existence of a logic circuit implementation. A consistent SG satisfies the CSC property if for every pair of states with the same binary codes the set of output events enabled in both states is the same. Pairs of states s, s' that violate the CSC condition are said to be in state encoding conflict or CSC *conflict* (binary codes 100*0 and 10*00 in Fig. 2(d)). In order to resolve CSC conflicts new state signals must be introduced in the specification [7], [10].

If these conditions are satisfied one can produce an SI circuit out of an STG in which each signal a will be implemented in $a = S + \overline{R} * a$ form, where R and S are set and reset gate functions respectively. This way of implementation is known as generalized C-element implementation (or gC-implementation simply).

3. Behavioral Approach for Delay-Insensitive Interfacing

Our approach has two distinctive features:

- It is focused not on *total* delay-insensitivity but on *delay-insensitive interfacing* only. The basic assumption is that within a module the designer or a 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 the module.
- Contrary to conventional approaches to DI synthesis, the tasks of designing a module and its environment are considered separately. This results in asymmetric DI interfacing requirements: only inputs are required to be *accepted* in a delay-insensitive fashion by the circuit, because delay-insensitivity with respect to outputs matters only when the implementation for the environment is synthesized.

The above conditions lead to a more relaxed axiomatic definition of *delay-insensitive interfacing* with respect to the classical definition of *delay insensitivity* given in [2]. A specification satisfies the *delay-insensitive interfacing requirement* if it meets the following conditions:

1. *No auto-concurrency.*
2. *Alternating inputs* (input events cannot be ordered with other input events).
3. *No cross-disabling* (inputs and outputs cannot disable each other).

Implication of these conditions in restricting the

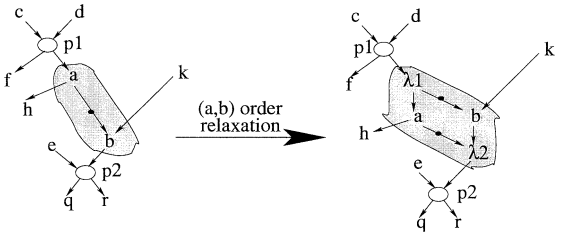

Fig. 4 Order relaxation.

"types" of STG with DI interfacing is given by Proposition 3.1. It was proved in [11].

Proposition 3.1: A consistent and output persistent STG satisfies DI interfacing conditions if and only if no input transition triggers another input transition.

The proof is trivial: non-auto-concurrency is a necessary condition of STG consistency, absence of cross-disabling is guaranteed by output persistency and alternation of inputs directly comes from the definition of DI interfacing.

Proposition 3.1 gives an idea about the places where DI interfacing might be violated in an STG: these are STG fragments in which input transitions are directly causally related. The addition of arbitrary delays to every input wire may unpredictably alter the order of originally ordered inputs to a module. This means that from the module point of view such inputs become concurrent. Hence the transformation of an STG for DI interfacing removes direct causal dependencies between inputs and makes them concurrent. This transformation can be performed by iterative application of a simple operation that is called *order relaxation* and is illustrated in Fig. 4. Informally an order relaxation between events a and b, such that event a is connected with b by a causal arc ($a \rightarrow b$), results in removing the arc between a and b (a and b becomes concurrent) and keeping the ordering between all other events in STG.

The following two properties of order relaxation help to clarify the transformation towards DI interfacing. Their proofs can be found in [13].

Property 3.1: Order relaxation between events a and b preserves pairwise ordering relations between all events except for a and b.

From Property 3.1 follows that the order of applying order relaxation between different events is irrelevant for the resulting STG, i.e., these behavioral transformations are commutative [13].

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

If in the original STG two inputs are directly causally related, then DI interfacing can only be obtained by relaxing the ordering between them. Such a relaxation, by Property 3.2, does not cause any new

Input: STG $A = \langle E, F, m_0 \rangle$ (E - events, F - precedence relations, m_0 - initial marking)
Output: STG $A_{di} = \langle E, F', m_0' \rangle$ with DI interfacing

```
foreach input events a and b, a → b do
  /* order relaxation step */
    remove causal arc (a, b)
    /* ordering predecessors of a with b */
    foreach predecessor p of a do
      add arc p → b;
    /* ordering successors of b with a */
    foreach successor s of b do
      add arc a → s;
    /*modify m0 if necessary */
    foreach initially marked arc (p, a) do
      m0((p, b))+ = m0((p, a));
    foreach b → s with arc (b, s) initially marked do
      m0((a, s))+ = m0((b, s));
    if (a, b) is initially marked then
      foreach arc (p, b) do m0((p, b))+ = m0((a, b))
  if STG A becomes auto-concurrent then exit(failure);
endfor
```

Fig. 5 Algorithm for ensuring DI interfacing.

cross-disabling 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. If non-auto-concurrency is preserved, the above transformation is *strictly delay-insensitivity increasing*. Its iterative application will eventually (if non-auto-concurrency is preserved) produce the new specification that will satisfy the requirements of DI interfacing.

The algorithm for STG transformation to ensure DI interfacing is presented in Fig. 5. The result of the algorithm is either a new STG, in which DI interfacing requirements are satisfied, or a *failure*, when input order relaxation leads to auto-concurrency. The latter implies that the original STG cannot be implemented with DI interface.

Example. Figure 6 illustrates the transformation leading to a DI interface for the *chu*133 benchmark example. (DI violations are depicted by shading). DI interfacing is achieved by iterative application of order relaxation between input events.

The order relaxation between $Lr-$ and $Zr+$ requires deleting arc $(Lr-, Zr+)$, adding direct predecessors of $Lr-$ to $Zr+$ (i.e. $Dr+ \rightarrow Zr+$) and adding direct successors of $Zr+$ to $Lr-$ (i.e. $Lr- \rightarrow Za+$). Similarly, the order between $Zr-$ and $Dr-$ is relaxed and the overall result is shown in Fig. 6(b). The new specification is non-autoconcurrent and has less DI violations than the original one.

Finally, the ordering between $Dr+$ and $Lr-$ is relaxed. This also preserves non-autoconcurrency and gives the desired specification with DI interface (Fig. 6(c)).

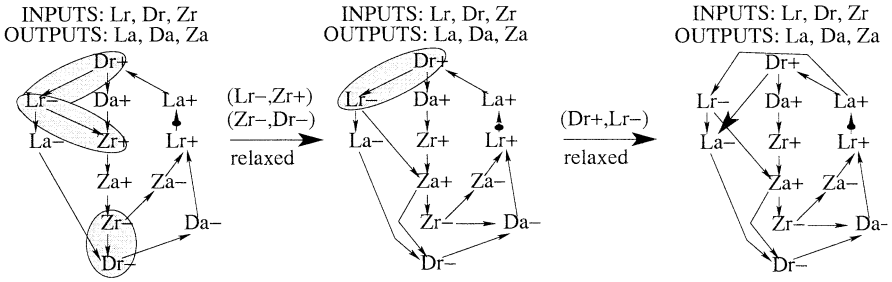

Fig. 6 Example of order relaxation.

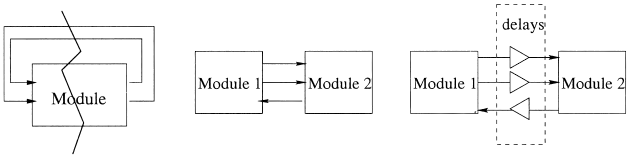

Fig. 7 The experimental flow for DI decomposition.

4. Experimental Results—Behavioral Approach

The experiment (illustrated in Fig. 7) started from a well-known asynchronous benchmark set with its environment. The set of signals was partitioned into two groups, yielding two separate modules as shown in Fig. 7(b). Each module plays the role of the environment for its counterpart, and the interface between them is made delay-insensitive by applying order relaxation between events which are input for each module. Note that this process does not always converge to a correct implementation because auto-concurrency may be created from order relaxation (this means that decomposition for DI interfacing could be used as a guidance criterion for asynchronous system partitioning even though there are several candidates to select a signal set, e.g., *chu*133(1) and *chu*133(2)). For all cases where DI interfacing could be obtained for some wire partition, we compared the DI implementation (Fig. 7(c)) against the SI one (Fig. 7(a)) in terms of area and performance. The results are shown in Table 1 (for area) and Table 2 (for performance) in the columns labeled SI and DI_{stg}. Area numbers are obtained by technology mapping into a virtual library using the Petrify tool. Performance is measured by using a Verilog simulator. On average the area penalty is about 36% and the performance degradation is about 20%.

The suggested approach was also applied for implementing a DI interface in a large scalable control circuit, whose STG specification had a regular structure. It originated from a practical case study of an asynchronous SI controller for an analog-to-digital converter (ADC) [14]. The results are in good correspondence with the penalty ranges obtained in the previous experiment—(38% for area and 7% for performance).

5. Gate-Level Approach for Delay-Insensitive Interfacing

The behavioral approach gives a global view on implications of the DI interface assumption for a circuit as a whole. It requires reconstructing its reachable state space, which grows due to the increased concurrency. The latter might pose computational difficulties in doing synthesis on the enlarged reachability space and systems that could be handled before might become unmanageable after the DI assumption has been made. It is very much likely that the order relaxation between a pair of inputs will impact only a small part of a circuit in a local vicinity of these inputs. This observation motivates our gate-level approach for ensuring DI interfacing [12]. It takes a SI circuit and locally transforms gates at the input boundaries (by adding literals) to satisfy DI interfacing.

Localizing DI violations. Suppose that a circuit C is a speed-independent implementation of STG A. Let us localize the gates of C that might be affected by replacing the SI assumption for input events $a*$ and $b*$, $a* \rightarrow b*$ with the DI one.

The most conservative approximation is given by a set of gates that are in the immediate fanout of both signals a and b. Indeed, under the unbounded gate delay model, none of the gates g for which either a or b is not input can be sensitive to the ordering of events on a and b because propagation delays of these transitions to g are arbitrary. For finer approximation let us distinguish the following cases:

- Set of gates G_{tr} that are triggered by event $b*$
- Set of gates G_{haz} in the fanout of both a and b that are not triggered by the firing of $b*$ in the original STG A but their outputs might have hazards when $a*$ and $b*$ are reordered

Property 5.1: Let C be a circuit obtained as a SI implementation from STG A in which $a*$ triggers $b*$ ($a* \rightarrow b*$) and A' be an STG obtained from A by deleting the causal arc (a, b) and ordering b with all the predecessors of a. If in the closed system created by circuit C and environment A'

1. gates G_{tr} switch only after the occurrence of $a*$ and
2. logic functions for gates G_{haz} do not change their values under any firing order of $a*$ and $b*$

then C satisfies DI interfacing for $a*$ and $b*$.

The proof of Property 5.1 is straightforward: Condition 1 tells that the gates from G_{tr} must propagate the impact of firing event $b*$ in the same way as before the order relaxation between $a*$ and $b*$ (no premature firings) while Condition 2 guarantees that there will be no additional activity in the circuit because of reordering $a*$ and $b*$ (no hazards either due to unexpected firings or lost excitations).

Information about the gates that are capable of premature firing can be easily obtained from the original STG A and circuit C: a gate g belongs to G_{tr} if $b* \rightarrow g*$ in A.

To estimate the set G_{haz} let us assume that the set of additionally reachable states (coming from the order relaxation between $a*$ and $b*$) is known and defined by characteristic function c_{add}. A gate might experience a hazardous behavior only if it is sensitized by b within c_{add} (otherwise the reordering of $a*$ and $b*$ does not influence it). For a gC-implementation ($g = S + \overline{R}g$) one can derive a necessary condition for a gate to be hazardous:

$$c_{add} * (\overline{g} * \delta S/\delta b + g * \delta R/\delta b) \neq 0 \quad (1)$$

Indeed, the term $\overline{g} * \delta S/\delta b$ (where $\delta S/\delta b$ denotes the boolean difference of S with respect to b) gives the set of states where the value of the logic function for g changes from 0 to 1 due to the firing of $b*$ while the term $g * \delta R/\delta$ characterizes its changes from 1 to 0. Finally we arrive to the following procedure of deriving the set of gates G_{haz}:

1. Choose a set of gates G in the immediate fanout of both a and b.
2. Exclude from G gates from G_{tr} (G_{haz} must not be triggered by $b*$ in STG A).
3. Exclude from G the gates not satisfying Condition (1) (they cannot have hazards due to reordering $a*$ and $b*$).

Constraining premature firings and hazards. Property 5.1 suggests a uniform way in treating gates from G_{tr} and G_{haz} when ensuring DI interfacing. Their logic functions must be kept constant in all additionally reachable states coming from reordering $a*$ and $b*$ (c_{add}). Let us first assume that the characteristic function c_{add} is known.

Then the following refinement procedure might be applied for each gate from G_{tr} and G_{haz}.

1. Calculate the set of DI conflicting states for rising transitions of g as $Conflict(g+) = c_{add} * \overline{g} * \delta S/\delta b$ (gate g might have either unspecified rising transitions or loss of the excitation for function S in these states).
2. Find the set of DI conflicting cubes for set function S of gate g as the ones that intersect with $Conflict/b$, where $Conflict/b$ is obtained from $Conflict$ by dropping literals of signals b.
3. For each DI conflicting cube c_{conf} find its cover refinement, if no refinement exists then return with a failure.
   a) When c_{conf} and $Conflict$ have the same value of signal b, gate g could have an unexpected firing in c_{add}. The set function must be restricted from hitting states in c_{add} (by choosing a signal that has opposite values in the states covered by c_{conf} and $Conflict$ e.g.).
   b) When c_{conf} and $Conflict$ have different values of signal b, the firing of $b*$ before $a*$ results in exiting ER($g+$) and entering states in c_{add} without the firing of gate g. This way of loosing excitation by g is improper. Restrict c_{conf} (if possible) by choosing a signal that has opposite values in the states covered by c_{conf} and $Conflict/b$.
4. Repeat the procedure for the set of states $Conflict(g-) = c_{add} * g * \delta R/\delta b$ and reset function R of the gate g.

If the above procedure terminates without failure then the modified set and reset functions of gate g do not intersect DI conflicting states and premature firings and hazards of g are blocked. In that way gates are modified to satisfy Conditions 1 and 2 of Property 5.1.

When a procedure fails (due to the lack of refinement or appearance of CSC conflicts in the reachability space after the order relaxation) the new signals should be added in STG in a very same way as in the framework of [7] (this topic is beyond the paper scope however).

Evaluation of the set of additionally reachable states. The minimal set of states coming from possible reordering between $a*$ and $b*$ is obtained when none of the successors of $b*$ fires before $a*$. The latter corresponds to the order relaxation between $a*$ and $b*$ based on minimal concurrency (like in Sect. 3). Assume for simplicity that $b* = b+$. Then the set of additionally reachable states could be evaluated by a cube $c_{add} = b * c_{min}$, where c_{min} is obtained by 1) taking any minterm from ER($b+$) and deleting in it literals of signals corresponding to events in $Conc(b+)$ (a set of events concurrent to $b+$) and 2) inverting the value of a if none of signal a events are in $Conc(b+)$ (due to the reordering $b+$ fires before $a*$ in c_{add}). In that way all the signals whose events are in $Conc(b+)$ are assumed to have arbitrary values during $b+$ firing and hence the cube c_{add} covers all additionally reachable states.

Example. Let us illustrate the suggested approach by constructing a DI interface for the chu150 benchmark STG (see Fig. 8(a)). The SI implementation based on generalized C-elements is shown in Fig. 8(b). There are

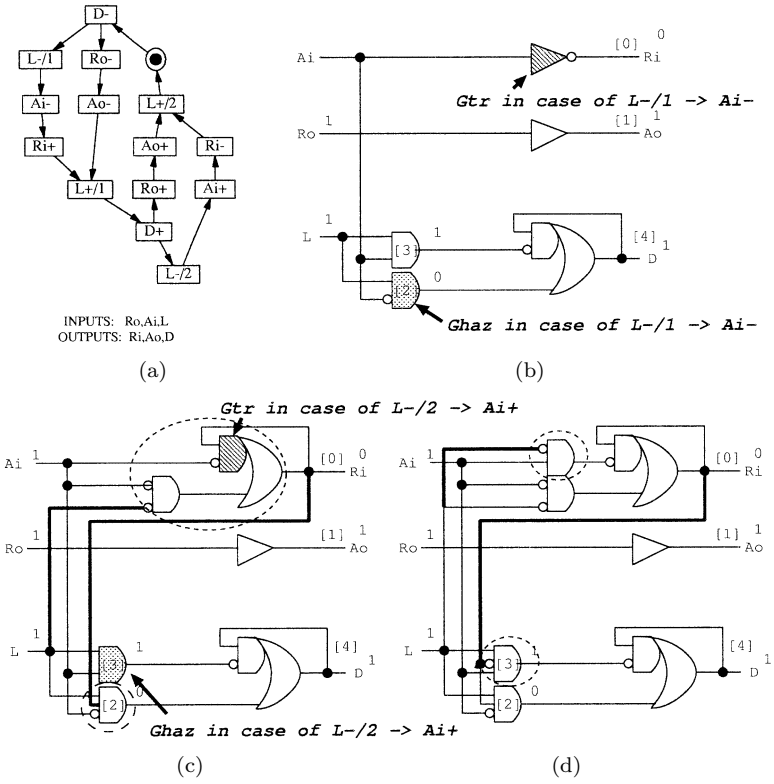

Fig. 8 Reduction to DI interfacing for *chu*150 example.

two cases of DI interface violations in the STG due to direct precedence of input events: 1) $L–/1 \rightarrow Ai-$ and 2) $L–/2 \rightarrow Ai+$. Firstly we consider case 1) $L–/1 \rightarrow Ai-$.

Case 1. Estimation of additionally reachable states due to reordering of $L–/1$ and $Ai-$. $Ai-$ is concurrent to $Ro-$ and $Ao-$. Therefore values of signals Ro and Ao are considered to be arbitrary in the cover cube c_{add}. The rest of signals keep their values in ER$(Ai-)$ $(L = 0, Ri = 0, D = 0)$. For the minimal concurrency in the order relaxation of $L–/1$ and $Ai-$ the cover cube c_{add} is approximated as $c_{add} = L * \overline{Ai} * \overline{Ri} * \overline{D}$.

Case 2. Constraining premature firings. $Ai-$ triggers a single event $Ri+$. Hence Ri is the only gate that belong to G_{tr}. For gate $Ri = \overline{Ai}$, implemented as an inverter, S and R functions are $\overline{Ai}$ and Ai respectively. Only S intersects with c_{add} and needs to be restricted. To delay the firing of $Ri+$ until $L–/1$ occurs, the function S is modified to $S = \overline{Ai} * \overline{L}$. This results in a modification of gate Ri to $Ri = \overline{Ai} * \overline{L} + Ri * \overline{Ai}$ (see Fig. 8(c)).

Case 3. Constraining hazards. The only gate in the fanout of L and Ai that is not triggered by $Ai-$ is $D = \overline{Ai} * L + D * \overline{L * Ai}$. The intersection of the set function for D $(S = \overline{Ai} * L)$ with the cube c_{add} is non-empty and S is sensitized by Ai. Due to this, D might have an unexpected positive transition when $Ai-$ and $L–/1$ arrive in reverse order. To restrict the set function of D from intersecting with c_{add} one can choose any variable that has opposite values in the states of ER$(D+)$ (covered by set function $S = \overline{Ai} * L$) and c_{add}. The only possible candidate is Ri $(S = \overline{Ai} * L * Ri$, $c_{add} = L * \overline{Ai} * \overline{Ri} * \overline{D}$ and $S \cup c_{add} = \emptyset)$. The refinement of S is shown by bold wire in Fig. 8(c).

After modifications the implementation (Fig. 8(c)) behaves properly under any order of occurrence for $Ai-$ and $L–/1$.

Similar consideration for the DI violation of Case 2 results in the following transformations:

1. The reset function of gate Ri $(Ri \in G_{tr}$ for $Ai+)$ is modified from $R = Ai$ to $R = Ai * \overline{L}$ (to delay firing Ri until $L–/2$ has occurred).

2. The reset function of gate D $(D \in G_{haz})$ is modified from $R = L * Ai$ to $R = L * Ai * \overline{Ri}$ (to avoid intersection with a cover cube $c_{add} = L * Ai * Ri * D$ for additional states and keep gate D stable during $Ai+$ and $L–/2$ transitions).

The final implementation is shown in Fig. 8(d).

6. Experimental Results—Gate-Level Approach

In order to estimate a quality of the suggested gate-level approach for DI transformations we made a comparison of its results with the implementations obtained through behavioral transformations. The comparison was done for both area and performance [12].

Table 1 gives the comparison of area cost for the

Table 1 Area comparison (# of literals).

name	SI (I)	DI_{stg} (II)	DI_{gate} (III)	ratio1 (II)/(I)	ratio2 (III)/(I)
chu133(1)	19	25	23	1.32	1.21
chu133(2)	19	23	21	1.21	1.11
chu150	28	30	28	1.07	1.00
mmu(1)	40	64	55	1.6	1.37
mmu(2)	40	54	47	1.35	1.18
mr	42	54	48	1.29	1.14
mr0	66	69	68	1.05	1.03
mr1	46	55	49	1.20	1.07
trimos	27	51	42	1.89	1.56
vbe10b	37	49	39	1.32	1.05
wrdatab	26	42	42	1.62	1.62
total	390	516	462	1.32	1.18

Table 2 Performance comparison (ns).

name	SI (I)	DI_{stg} (II)	DI_{gate} (III)	ratio1 (II)/(I)	ratio2 (III)/(I)
chu133(1)	5147	6730	6936	1.30	1.35
chu133(2)	5147	6636	6829	1.29	1.33
chu150	8860	9469	8860	1.07	1.00
mmu(1)	6118	8193	9495	1.34	1.55
mmu(2)	6118	7915	8840	1.29	1.44
mr	7328	7188	80956	0.98	1.10
mr0	11117	12574	12746	1.13	1.15
mr1	9182	11308	9476	1.23	1.03
trimos	1844	7856	3477	4.26	1.89
vbe10b	9757	12187	10515	1.25	1.08
wrdatab	8003	9989	11219	1.25	1.40
total	78621	100045	96488	1.27	1.23

two DI interfacing approaches. Column SI shows the number of literals in the gC-implementation for the original SI circuit, while DI_{stg} and DI_{gate} show the literal count for the circuits with DI interfacing obtained via behavioral and gate-level transformations respectively.

Table 2 provides similar data for a performance metric.

The results of Table 1 show that the gate-level approach often outperforms the behavioral approach in providing less area overhead for ensuring DI interfacing. The reasons behind that are as follows:

- The gate level approach is based on relatively cheap modifications of gate functions, usually a modification simply adds a literal (wire) to a gate. The modifications are also of local nature because they involve only the gates in the immediate fanouts of reordered signals.
- Behavioral transformations give a global view on the DI interfacing problem. However, because of the huge solution space involved heuristics have to be applied to prune it out. The amount of concurrency in transformations is one of such heuristics. If one could explore all concurrency possibilities, then the results of behavioral method would of course be optimal. Unfortunately, the way of exploring concurrency is far from perfect, which may affect the quality of implementation in comparison to other heuristic approaches.

Data on performance comparison are less consistent and show that both approaches give similar performance penalties for DI interfacing.

Although Table 1 illustrates that application of the gate-level approach to ensure DI interfacing might be encouraging it is worth to mention several difficulties met on that way:

- The approach works well only for simple cases, when going from SI to DI assumptions does not produce CSC conflicts in the expanded reachability space.
- For the moment the approach gives a clear path only for circuits implemented by generalized C-elements.
- The "level of service" delivered in gate-level methods is lower because they are not fully automated (contrary to the behavioral approach).

Summarizing these observations, the following design scenario might be suggested. At first behavioral transformations are applied to derive implementations with DI interfacing. If they fail (due to high computational complexity) the simpler gate-level transformations might be applied. Note that currently the application of both transformations is restricted to the gC-based implementations.

7. Conclusions

Wire delays draw special attention in moving to deep submicron technologies. Design styles which neglect wire delays seem to be overly optimistic even with the 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.

This paper suggests a design methodology which can tolerate skew in interface signals. In this methodology a designer identifies a set of "dangerous" wires that are implemented in a delay-insensitive fashion, while for the rest of a circuit other (more conventional) design styles are applied. In particular, we used speed-independent implementation for the parts of a system in which wire delays could be controlled by a designer or a routing tool, and then applied the delay-insensitive hypothesis only to the wires running between such speed-independent "islands."

Two different ways to ensure DI interfacing are developed. Their combination gives powerful means for exploring the optimization space of implementations with DI interface.

References

[1] T.A. Chu, Synthesis of self-timed VLSI circuits from graph-theoretic specifications, Ph.D. Thesis, MIT, 1987.
[2] J.T. Udding, "A formal model for defining and classifying delay-insensitive circuits and systems," Distributed Computing, vol.1, pp.197–204, 1986.
[3] R.H.J.M. Otten and R.K. Brayton, "Planning for performance," Proc. Design Automation Conference '98, June 1998.
[4] D. Sylvester and K. Keutzer, "Getting to the bottom of deep submicron," Proc. International Conference on Computer-Aided Design '98, Nov. 1998.
[5] C.L. Seitz, Introduction to VLSI systems, Chapter 7, Addison Wesley, 1981.
[6] D.M. Chapiro, Globally-asynchronous locally-synchronous systems, Ph.D. Thesis, Stanford University, 1984.
[7] 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. Circuits & Syst., vol.E80-D, no.3, pp.315–325, March 1997.
[8] T. Verhoeff, "Delay-insensitive codes—An overview," Distributed Computing, vol.3, no.1, pp.1–8, 1988.
[9] I.E. Sutherland, "Micropipelines—The 1988 Turing award lecture," Commun. ACM, vol.32, no.6, pp.720–738, June 1989.
[10] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man, "A generalized state assignment theory for transformations on signal transition graph," Proc. International Conference on Computer-Aided Design '92, pp.112–117, Nov. 1992.
[11] H. Saito, A. Kondratyev, J. Cortadella, L. Lavagno, and A. Yakovlev, "What is the cost of delay insensitivity?" Proc. International Conference on Computer-Aided Design '99, pp.316–323, Nov. 1999.
[12] H. Saito, A. Kondratyev, and T. Nanya, "Design of asynchronous controllers with delay insensitive interface," Proc. Asia South Pacific Design Automation Conference '02, pp.93–98, Jan. 2002.
[13] H. Saito, A. Kondratyev, J. Cortadella, L. Lavagno, and A. Yakovlev, "What is the cost of delay insensitivity?," Technical Report, 99-2-004, The University of Aizu, Aug. 1999.
[14] D.J. Kinniment, B. Gao, A.V. Yakovlev, and F. Xia. "Toward asynchronous A-D conversion," Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp.206–215, 1998.

Hiroshi Saito received the M.S. degree in computer science from the University of Aizu, Japan in 2000. Currently, he is a Ph.D. student of the Advanced Interdisciplinary Studies Engineering of the University of Tokyo, Japan. His research interest includes synthesis of asynchronous circuits.

Alex Kondratyev received the M.S. (1983) and Ph.D. (1987) degrees in computer science from the Electrotechnical University of St.Petersburg, Russia. Since 2001 he is a research scientist in Cadence Berkeley Laboratories. His research interests include synthesis of asynchronous circuits, theory of concurrency and system design.

Jordi Cortadella received the M.S. and Ph.D. degrees in Computer Science from the Universitat Politecnica de Catalunya, Barcelona, Spain, in 1985 and 1987 respectively. He is a professor in the Department of Software of the same university. 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.

Luciano Lavagno graduated in 1993 from Politecnico di Torino and received his Ph.D. from the University of California at Berkeley in 1992. Since 2001 he is an Associate Professor with the Department of Electronics of Politecnico di Torino, Italy. His research interests include synthesis and testing of asynchronous circuits, and the design of mixed hardware and software embedded systems.

Alex Yakovlev received the M.Sc. (1970) and Ph.D. (1982) degrees in computer science from the Electrotechnical University of St.Petersburg. Since 2000 he is a professor at the Newcastle University Computing Science Department. His publications and research interests are in modeling and design of asynchronous, concurrent and real-time systems.

Takashi Nanya received the B.E. and M.E. degrees in Mathematical Eng. and Information Physics from the University of Tokyo, Japan, in 1969 and 1971, respectively, and the Ph.D. degree in Electrical Eng. from the Tokyo Institute of Technology, Japan in 1978. In 1996, he joined the University of Tokyo, where he is a professor at Research Center for Advanced Science and Technology. His research interests include fault-tolerant computing.