Technology Mapping for Speed-Independent Circuits: Decomposition and Resynthesis

Alex Kondratyev, The University of Aizu, Japan
Jordi Cortadella, Univ. Politecnica de Catalunya, Barcelona, Spain*
Michael Kishinevsky, The University of Aizu, Japan
Luciano Lavagno, Politecnico di Torino, Italy †
Alex Yakovlev, University of Newcastle upon Tyne, United Kingdom ‡

Abstract

This paper presents theory and practical implementation of a method for multi-level logic synthesis of speed-independent circuits. An initial circuit implementation is assumed to satisfy the monotonous cover conditions but is technology independent. The proposed method performs both combinational (inserting new gates) and sequential (inserting new memory elements) decomposition of complex gates in a given standard cell library, while preserving original behaviour and speed-independence. The algorithm applies known efficient algebraic factorization techniques from combinational multi-level logic synthesis, but achieves also boolean *simplification and sequential decomposition. The method allows sharing of decomposed logic.*

1 Introduction

Speed-independent circuits, originating from D.E. Muller's work [11], are *hazard-free under the unbounded gate delay model.* With recent progress in developing efficient analysis and synthesis techniques, supported by CAD tools, this sub-class has moved closer to practice, bearing in mind the advantages of speed-independent designs, such as their greater temporal robustness and self-checking properties.

Existing methods of logic synthesis for speed-independent circuits either assume that the implementation library contains *and* gates with unbounded fanin and "free" input inversions ([1, 5, 9]) or they use non-standard "hazard absorbing" flip-flops whose effectiveness *in practice* still needs to be evaluated ([14]). Other results on the implementability of semi-modular circuits without inputs using two-input/two-output *and* and *or* gates ([18]) are only interesting from a theoretical standpoint, due to their extremely high implementation cost.

In attempts to map speed-independent circuits into a more realistic, standard cell-like, library, other sort of restrictions have been exercised. For example, the approach described in [16] works only under the *fundamental mode assumption*, which is overly restrictive and does not fit well theoretically with the unbounded delay assumption. The same authors describe in [15] a method to perform technology mapping for speed-independent circuits that only decomposes existing gates (e.g., a 3-input *AND* into two 2-input *AND*s), without any further search of the implementation space. They do not explore complex decompositions, that could use multi-cube divisors, or decompose several gates simultaneously. The same limitations also affect the work of [1, 2]. The idea of complete resynthesis of a circuit every time a new signal is inserted is exploited in [12] for the technology mapping of timed asynchronous circuits. However the search space for decomposition is again limited by a single signal network.

In [13] a method for technology mapping of speed-independent circuits using complex gates was presented. This method however only identifies when a set of simple logic gates can be implemented as a complex gate, but cannot perform a speed-independent decomposition of a signal function in case it does not fit into a single gate. In fact, this method can be used as a post-optimization step after our proposed decomposition technique.

Finally, Burns analyzes [4] the correctness conditions for a decomposition of a sequential element that is part of a speed-independent circuit into two sequential elements (or a sequential and a combinational element). Notably, these conditions are analyzed using the original (unexpanded) behavioural model, thus helping the efficiency of the method. This work is, in our opinion, a big step in the right direction, but addresses mainly correctness issues. It does not describe how to use the efficient correctness checks in an optimization loop, and does not allow the sharing of a decomposed gate by different signal networks.

The idea of combinational logic decomposition with resynthesis has been proposed in [8, 7]. The approach combines together efficient algebraic factorization techniques used in multi-level combinational logic synthesis (finding candidates for decomposition), and speed-independence preserving signal insertion (the latter idea originated in [17] and was implemented efficiently in [6]).

The main contribution of this paper is a generalisation and extension of the above basic idea so as to cover both combinational and sequential decomposition. We have

*This work has been partly supported by the Ministry of Education of Spain (CICYT TIC 95-0419), ACiD-WG (ESPRIT 21949) and integrated action UK HB1995-0203.

†This work has been partly supported by MURST research project "VLSI architectures".

‡Work supported by UK EPSRC GR/L24038, ACiD-WG (ESPRIT 21949) and British Council integrated action Spain (MDR/1996/97/1159)

developed a body of theory that allows us to prune the search space when looking for solutions. We continue to use classical logic synthesis techniques available for combinational multi-level logic in order to find good candidate functions for the decomposition. In the case of combinational decomposition the newly inserted signal is a library gate. The insertion of a combinational gate is based primarily on one of the two transitions of the gate's output (e.g., its rising transition). The other transition of the combinational gate is fully determined by the insertion place of the first transition.

A sequential decomposition, based on a new memory element, can improve the progress of mapping by rendering the opposite transition a more effective role, since the set and reset logic are inserted independently. In particular, two boolean functions can be decomposed at the same time with one new signal. Thus, in comparison with [4], this method:

- targets the search of the solution towards a given library;
- allows logic sharing based on multiple acknowledgments;
- performs global optimization via resynthesis (rather than sequential decomposition).

Throughout the paper we use the following notation:

- A stands for the original State Graph, A' -- for a new State Graph obtained by signal insertion.
- $a, b, c, \ldots$ (lower case Latin letters) are used for signal names and, corresponding to them, literals in Boolean functions.
- x - always denotes a *new* signal, which is inserted in State Graph A to decompose a non-implementable function.
- $B, C, F, P, Q, R, \ldots$ (upper case Latin letters, except A) stand for the names of Boolean functions.

2 Theoretical background

In this section we introduce theoretical concepts required for our decomposition method: (1) circuit specification and its logic implementability; (2) conditions for speed-independent decomposition of complex gates; and (3) transformations of state graphs to ensure those conditions.

2.1 State Graphs and Logic Implementability

A *State Graph* (*SG*) is a labeled directed graph whose nodes are called *states*. Each arc of an *SG* is labeled with an *event*, that is a rising ($a+$) or falling ($a-$) transition of a signal a in the specified circuit. We also allow notation $a*$ if we are not specific about the direction of the signal transition. Each state is labeled with a vector of signal values. An *SG* is *consistent* if its state labeling $v : S \rightarrow \{0,1\}^n$ is such that: in every transition sequence from the initial state, rising and falling transitions alternate for each signal. Figure 1,b shows the *SG* for the Signal Transition Graph in Figure 1,a, which is consistent. We write $s \xrightarrow{a}$ ($s \xrightarrow{a} s'$) if there is an arc from state s (to state s') labeled with a.

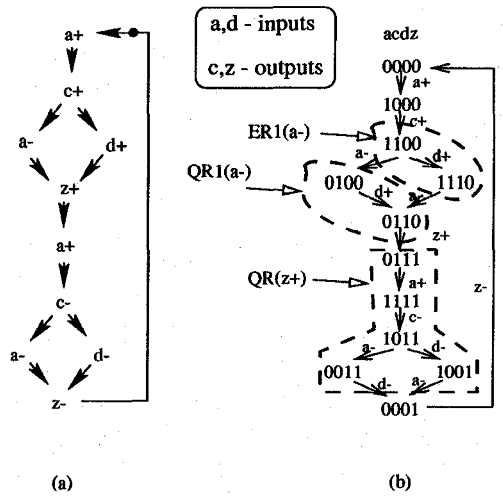

Figure 1: An example of State Transition Graph (a) and State Graph (b) (benchmark *hazard.g*)

The set of all signals whose transitions label *SG* arcs are partitioned into a (possibly empty) set of inputs, which come from the environment, and a set of outputs or state signals that must be implemented. In addition to consistency, the following two properties of a *SG* are needed for their implementability in a speed-independent logic circuit.

The first property is *speed-independence*. It consists of three constituents: determinism, commutativity and output-persistency. A *SG* is called *deterministic* if for each state s and each label a there can be at most one state s' such that $s \xrightarrow{a} s'$. A *SG* is called *commutative* if whenever two transitions can be executed from some state in any order, then their execution always leads to the same state, regardless of the order. An event a^* is called persistent in state s if it is enabled at s and remains enabled in any other state reachable from s by firing another event b^*. A *SG* is called *output-persistent* if its output signal events are persistent in all states. Any transformation (e.g., insertion of new signals for decomposition), if performed at the *SG* level, may affect all three properties.

The second property, *Complete State Coding* (*CSC*), becomes necessary and sufficient for the existence of a logic circuit implementation. A consistent *SG* satisfies the *CSC* property if for every pair of states s, s' such that $v(s) = v(s')$, the set of output events enabled in both states is the same. (The *SG* in Figure 1,b is *output-persistent* and has *CSC*.) *CSC* does not however restrict the type of logic function implementing each signal. It requires that each signal is cast into a *single atomic* gate. The complexity of such a gate can however go beyond that provided in a concrete library or technology.

2.2 Gate-level implementability without hazards

Necessary and sufficient conditions for speed-independent implementation using unbounded fanin *and* gates (with unlimited input inversions), bounded fanin *or* gates and *C* elements were given in [1, 9]. In this work we are considering a similar basic implementation architecture, called the *standard-C* architecture, which is described in Figure 2. The difference from previous work is that instead of unbounded fanin gates for the set and reset logic of C-elements, we will allow only *implementable* gates, that is the gates which exist in the chosen library.

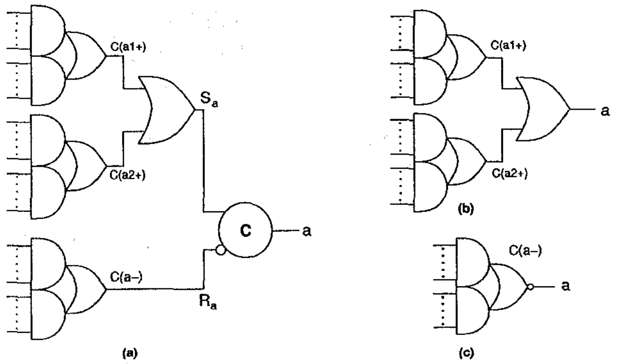

Figure 2: The standard-C architecture extended for complex gates

The concepts of excitation and quiescent regions are essential for that. A set of states is called an *excitation region* (*ER*) for event a^* (denoted by $ER_j(a^*)$) if it is a *maximal connected* set of states such that $\forall s \in ER_j(a^*)$: $s \xrightarrow{a^*}$. Since any event a^* can have several separated *ER*s, an index j is used for the distinction between *different connected occurrences* of a^* in the *SG*.

The *quiescent region* (*QR*) (denoted by $QR_j(a^*)$) of a transition a^*, with excitation region $ER_j(a^*)$, is a *maximal* set of states s reachable from $ER_j(a^*)$ such that a is stable in s and s is not reachable from any other $ER_k(a^*)$ such that $k \neq j$ without going through $ER_j(a^*)$ [1]. Examples of *ER* and *QR* are shown in Figure 1,b.

Let $C_j(a^*)$ denote one of the first-level AND-OR gates in the standard-C architecture. $C_j(a^*)$ is a *correct monotonous poly-term cover*[2] for the excitation region $ER_j(a^*)$ if the following three conditions are satisfied:

1. *Cover condition:* $C_j(a^*)$ covers all states of $ER_j(a^*)$ (i.e., $C_j(a^*)$ evaluates to 1 in all states of $ER_j(a^*)$).

2. *One-hot condition:* $C_j(a*)$ does not cover any state outside $ER_j(a*) \cup QR_j(a*)$.

3. *Monotonicity condition:* $C_j(a^*)$ changes at most once along any state sequence within $QR_j(a^*)$.

The conditions above are called the *Monotonous Cover conditions* or shortly the *MC-conditions.* Since under these conditions the outputs of the first-level gates are *one-hot encoded* any valid Boolean decomposition of the second-level *or* gates is speed-independent.

The standard-C architecture permits a *combinational* implementation of a signal. If the set and reset networks are the complements of each other, then a C-element with identical inputs can be simplified to a wire (see Figure 2,b,c). In such case we say that the signal has a *complete* cover.

2.3 Property-preserving event insertion

Our decomposition method is essentially behavioural -- the extraction of new signals at the structural (logic) level must be matched by an insertion of their transitions at the behavioural (*SG*) level. Event insertion is an operation on a *SG* which selects a subset of states, splits each of them into two states and creates, on the basis of these new states, an excitation region for a new event. Figure 3 shows the chosen insertion scheme, analogous to that used by most authors in the area [17].

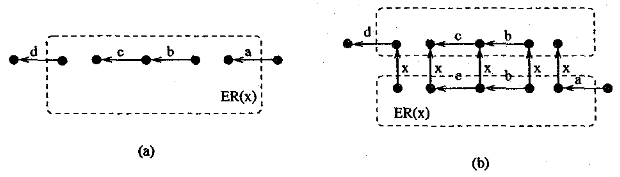

Figure 3: Event insertion scheme: (a) before insertion, (b) after insertion

State signal insertion must preserve the *speed-independence* of the original specification. An inserted signal is denoted by x in this paper. The corresponding to it events are denoted $x*$, $x+$, $x-$, or, if no confusion occurs, simply by x. Let A be a *SG* and A' is a state graph obtained by insertion of event x. We say that an insertion state set $ER(x)$, in a *SG A* is a *speed-independence preserving set (SIP-set)* iff: (1) for each event a in A, if a is persistent in A, then it remains persistent in A', and (2) A' is deterministic and commutative. The formal conditions for the set of states r to be a SIP-set can be given in terms of intersections of r with the so-called state diamonds of SG [6]. These conditions are illustrated by Figure 4, where all possible cases of the illegal intersections of r with state diamonds are shown.

It was shown in [6] that the insertion of a signal by means of a SIP-set is a necessary and sufficient condition to preserve the speed-independence of a corresponding SG. This requirement is the most general one in the synthesis of speed-independent circuits and it does not restrict the solution space unless we go beyond the speed-independent class. An efficient method for finding SIP-sets, which is based on regions, has been proposed in [6]. The first method for finding SIP-sets based on reduction to satisfiability problem was proposed in [17].

Assume that the set of states S in a *SG* is partitioned into two subsets which are to be encoded by means of an additional signal. This new signal can be added either in order to satisfy the *CSC* condition, or to break up a complex gate into a set of smaller gates. In the latter case, a new signal represents the output of the intermediate gate added to the circuit. Let r and $\overline{r} = S - r$ denote the blocks of such a partition. For implementing such a partition we

[1]Note that contrary to [9, 1] in this paper we use only the so-called restricted quiescent regions which do not include states reachable directly from two different excitation regions of the same signal.

[2]Here for simplicity we consider the definition of Monotonous Cover without the extension by the so-called *backward quiescent regions* and without considering covering of multiple regions by the same cover. However all the results can be easily generalized for this extension as well.

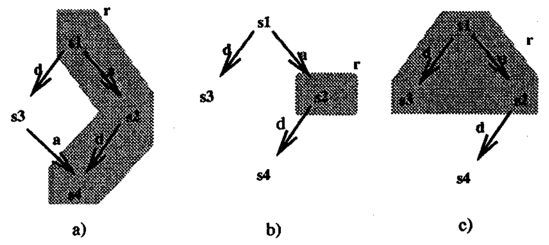

Figure 4: Possible violations of SIP conditions

need to insert transitions of the new signals in the *border states* between r and $\overline{r}$.

In this paper we shall consider the so-called *input border* of a partition block r, denoted by $IB(r)$, which is informally a subset of states of r by which r is entered. We call $IB(r)$ *well-formed* if there are no arcs leading from states in $r - IB(r)$ to states in $IB(r)$. If a new signal is inserted using an input border, which is not well-formed, then the consistency property is violated. Therefore, if an input border is not well-formed, its well-formed speed-independent preserving closure is constructed, as described by Algorithm 4.1 in Section 4.

The insertion of a new signal can be formalized with the notion of *I-partition* ([17] used a similar definition). Given a *SG*, A, with a set of states S, an I-partition is a partition of S into four blocks: $\{S^+, S^1, S^-, S^0\}$. $S^0(S^1)$ defines the states in which x will have the stable value 0 (1). $S^+(S^-)$ defines $ER(x+)$ $(ER(x-))$ in the new *SG* A'. Therefore, abusing notation we will often refer to $S^+(S^-)$ as to $ER(x+)$ $(ER(x-))$ when talking about states of the *original SG* A or, if confusion may arise, we write $ER_A(x+)$ $(ER_A(x-))$. If the insertion of x preserves consistency and persistency, then the only transitions crossing boundaries of the blocks are the following: $S^0 \rightarrow S^+ \rightarrow S^1 \rightarrow S^- \rightarrow S^0$.

3 Decomposition techniques

We assume here familiarity with multi-level logic synthesis (see [3] for more details).

As described in the previous section, any deterministic, commutative, output-persistent *SG* satisfying the *CSC* and the Monotonous Cover conditions can be implemented using the standard-C architecture. We assume that C-elements are present in the library [3]. OR-gates combining cover functions $C(a*)$ can be decomposed by any standard technique since their inputs are one-hot encoded. Hence the bottleneck for technology mapping is the implementation of cover functions $C(a*)$ using gates available in the library.

As traditionally done in multi-level combinational synthesis, we have chosen algebraic division as the main operation for logic decomposition. Thus, for each cover function $C(a*)$ we seek algebraic divisors, aiming at decompositions of the following type: $C(a*) = F * G + R$ where G is the quotient $C(a*)/F$. AND-decomposition is done when $R = 0$, whereas OR decomposition occurs when $G = 1$.

[3]In fact our technique works and is implemented also for RS- and D-latches. However, this generalization of the method is omitted due to the lack of space.

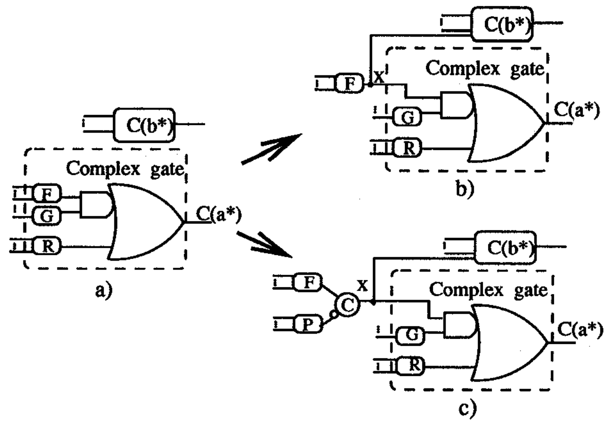

Figure 5: Cover function $C(a*)$ (a) and its combinational (b) and sequential (c) decompositions

However, contrary to the classical combinational decomposition we use divisor F not for immediate extraction, but as a first approximation of the function to be extracted. More specifically, function F defines one (sequential decomposition) or two (combinational decomposition) blocks of a partition of the state space, which is later used for new signal insertion (see Sections 4 and 5 for more details).

Two ways of decomposing $C(a*)$ are possible:

- combinational decomposition: a divisor F is implemented by a combinational gate, x, as shown in Figure 5,b and
- sequential decomposition: an additional latch (e.g., C-element) implements signal x; divisor F is used as one of the input functions for the latch as shown in Figure 5,c. Another function (denoted by P in the figure) must be extracted from some other cover function. Functions F and P form the set and reset functions for the new sequential signal x.

In our decomposition technique transitions of x are acknowledged by *several* cover functions. This is more general and powerful than [15, 4] where transitions of x must be acknowledged locally, only by the cover function $C(a*)$ from which x is extracted. Multiple acknowledgment offers two advantages: (1) the same signal x can be shared by several cover functions (this corresponds to the extraction of common sub-dividers in classical multi-level decomposition) and (2) correct speed-independent decomposition can be found even if it does not exist for solutions with single acknowledgments (see the experimental results). Note that we do not specifically search for multiple acknowledgments. They appear automatically due to the signal insertion technique based on SIP-sets. Hence our solution is correct by construction and contrary to [2] never requires iterations with verification procedures.

To find good divisors F for $C(a*)$ the following functions are considered:

- Kernels and co-kernels of $C(a*)$.

• If $C(a*)$ is a poly-term cover, any subset of terms of the sum-of-product expression (OR-decomposition).

• If $C(a*)$ is one cube, any subset of literals of the cube (AND-decomposition).

• Recursive decomposition of the previous candidates, e.g. sub-kernels and AND/OR-decomposition of kernels.

This generation of divisors is heuristically pruned to avoid an explosion of candidates for functions with many terms or cubes with many literals. Experimental results (Section 6) have shown this type of decomposition to be very effective. In particular, only those decompositions are considered that:
(1) preserve speed-independence and
(2) guarantee progress in mapping the circuit to the given library.

The first condition is satisfied by finding an I-partition for signal x. Many candidates for decomposition are filtered out at this step, since for many divisors there are no valid I-partitions.

To clarify the second condition assume that function F is extracted from a cover function $C(a*)$ for combinational decomposition (see Figure 5,b). If there is a valid I-partition for a new signal x, then there is a speed-independent implementation for the circuit with signal x. However, in general, there is no guarantee that function $C(a*)$ is simplified in the new circuit. The substitution of x for F in $C(a*)$ does not always preserve speed-independence and hence new fan-in signals for $C(a*)$ can appear in the implementation. Thus, the progress condition checks whether a substitution of x instead of F in $C(a*)$ is valid.

Since multiple acknowledgment of x can appear, the requirement for "good decomposition" is following: the complexity of all (other than $C(a*)$) functions in x's fan-out has to remain the same or to increase very moderately. In Section 4.3 we present a computationally efficient method for the estimation of effective decompositions.

The overall algorithm for logic decomposition is sketched below. The next sections describe each step in more detail.

Algorithm 3.1 (Speed-independent decomposition)

```
while circuit is not mapped to the library do
  Calculate monotonous covers for all events;
  Let $a*$ be the event with the most complex cover;
  Let $D_a$ be a set of divisors for $C(a*)$;
  /* Kernels, co-kernels, AND/OR decomposition */
  Let $D_{\overline{a}}$ be a set of divisors for the most complex
  cover functions other than $C(a*)$;
  for each $F \in D_a$ do
    Decomposition($F, \overline{F}$) ;
    /* Check combinational decomposition for $F$ */
    for each $P \in D_{\overline{a}}$ do
      Decomposition($F, P$)
      /* Check sequential decomposition for the pair $\{F, P\}$ */
    end for
  end for
  if All decompositions fail then
    return; /* Cover $C(a*)$ cannot be decomposed */
  else
    Choose the best decomposition ($\{F, \overline{F}\}$ or $\{F, P\}$);
    Insert a new signal
    /* by an I-partition defined by the best decomposition */
  end if
end while

Decomposition($F, P$);
  Find I-partition for the pair $\{F, P\}$;
  if not exists then return failure;
  Evaluate progress for decomposition of $C(a*)$;
  /*(proposition 4.1) */
  if no progress then return failure;
  Estimate progress for all other covers;
  /* (property 4.5) */
  if implementability is disturbed then return failure
```

Note that after each cycle, when a successful decomposition is found, the implementation of *every signal* in the circuit is recomputed for the best candidate. Since at the recomputation step the new don't care sets are used for all signals, this practically implements sequential decomposition and *boolean* division (i.e., it is far beyond the capabilities of algebraic factorization).

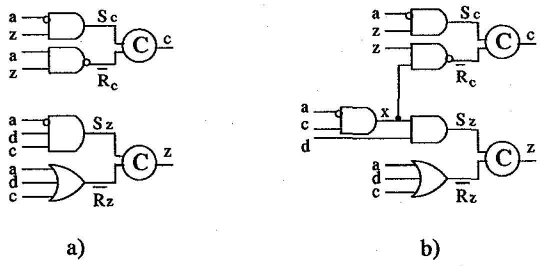

Figure 6: Circuits for a *hazard.g* example before (a) and after (b) decomposition

Example *hazard.g*. This example (from the set of asynchronous benchmarks) is used for illustrating our algorithm. Its Signal Transition Graph and *SG* are shown in Figure 1,a and b. Signals a and d are inputs, signals c and z -- outputs. A speed-independent implementation of the output signals c and z is presented in Figure 6,a. Our target is the decomposition of function S_z into two-input gates, because it is a standard worst case against which the performance of a decomposition algorithm can be measured. Function S_z consists of a single 3-literal cube $\overline{a}dc$. It can be decomposed in three ways: by extracting functions $\overline{a}d$, $\overline{a}c$ and dc.

Example 2. For the cover $C(y*) = ab+ac+def$ the following divisors are generated (trivial 1-literal divisors are not considered): the kernel $b + c$, the OR-decompositions ab, ac, def, $ab + ac$, $ab + def$ and $ac + def$ and the AND-decompositions de, df and ef.

4 Combinational decomposition

4.1 State partitioning

In this section we apply the theory of SIP-insertion, reviewed in Section 2.3, to a divisor F of a given cover $C(a*)$.

Definition 4.1 (Transition sets) *Let $A = < V, E >$ be a* SG *with a set of states V and a set of events E. Let $S \subseteq V$ be a subset of states and $e \in E$ be an event. The following sets of states are defined for S and e (see Figure 7):*

$$\begin{aligned} before(e,S) &= \{s : s \notin S \wedge \exists s'(s \xrightarrow{e} s' \wedge s' \in S)\} \\ entry(e,S) &= \{s : s \in S \wedge \exists s'(s' \xrightarrow{e} s \wedge s' \notin S)\} \\ leave(e,S) &= \{s : s \in S \wedge \exists s'(s \xrightarrow{e} s' \wedge s' \notin S)\} \\ after(e,S) &= \{s : s \notin S \wedge \exists s'(s' \xrightarrow{e} s \wedge s' \in S)\} \end{aligned}$$

$$\begin{aligned} Pred(S) &= \bigcup_{e \in E} before(e,S) \\ IB(S) &= \bigcup_{e \in E} entry(e,S) \\ EB(S) &= \bigcup_{e \in E} leave(e,S) \\ Succ(S) &= \bigcup_{e \in E} after(e,S) \end{aligned}$$

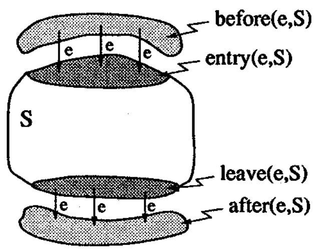

Figure 7: Illustration of transition sets

$Pred(S)$ and $Succ(S)$ give the sets of states outside S reachable in one step in backward or forward direction, respectively. Input, $IB(S)$, and exit, $EB(S)$, borders of S give the sets of states inside S from which the states not included in S are reachable in one step in backward or forward direction, respectively. Our technique operates with input borders. Set $IB(\overline{F})$ defined by Definition 4.1 can be computed as follows ($IB(F)$ is computed similarly):

$$IB(\overline{F}) = \bigcup_{e \in E} entry(e, \{s : F(s) = 0\}) =$$
$$= \{s : F(s) = 0 \wedge \exists s_1 : s_1 \rightarrow s \wedge F(s_1) = 1\}.$$

An event $b*$ is said to be a *trigger event* for event $a*$ if $entry(b*, ER(a*)) \neq \emptyset$. Informally, by firing trigger events it is possible to enter the excitation region for $a*$. We also say that signal b is a *trigger signal* for signal a and for event $a*$. All trigger signals for signal a must be included in the support of the logic function implementing a and hence each trigger signal will be in the fan-in of a. Triggers can be easily derived by observing ERs of a in the *SG*.

We can also show another property of trigger signals, that will be used to estimate the complexity of the logic after decomposition.

Property 4.1 *Event $x*$ is a trigger for event $b*$ in SG A' iff $leave(b*, ER(x*)) \neq \emptyset$.*

The proof follows directly from the rules for event insertion (cf. Figure 3), because if $leave(b*, ER(x*)) \neq \emptyset$ then the firing of $b*$ will be delayed until $x*$ has fired.

Any boolean function F defines a bipartition $\{S^F, S^{\overline{F}}\}$ of the set of states of a *SG*: $S^F = \{s : F(s) = 1\}$ and $S^{\overline{F}} = \{s : F(s) = 0\}$. As discussed in Section 2.3, for inserting a new signal x it is necessary to find an I-partition, $\{S^+, S^1, S^-, S^0\}$, based on bipartition $\{S^F, S^{\overline{F}}\}$. The four blocks of I-partition are constructed as follows:

- $S^- = ER(x-) \subseteq S^{\overline{F}}$ and $S^+ = ER(x+) \subseteq S^F$, corresponding to the excitation regions of x in the new *SG*, are obtained by the well-formed closure of the input border sets, $IB(\overline{F}) \subseteq S^{\overline{F}}$ and $IB(F) \subseteq S^F$, respectively [6].

- $S^1 = S^F - S^+$ and $S^0 = S^{\overline{F}} - S^-$.

The following property states that, if there is a well-formed SIP closure of the IB, then there is a minimal closure that has strictly less states than any other.

Property 4.2 *[8] Let $\{b, \overline{b}\}$ be a bipartition of the* SG *states. Let $I_1 \subseteq b$ and let I_2 be a minimal well-formed SIP set such that $I_1 \subseteq I_2 \subseteq b$. Then I_2 either does not exist or unique.*

In particular (the practically useful case), this property holds for $I_1 = IB(b)$. The proof (see [8]) provides a constructive procedure for selecting the minimal well-formed SIP closure of the input border without backtracking and thus is computationally efficient. This procedure can be summarized as follows. (We further illustrate it by deriving $ER(x-)$ for $F = dc$ in S_z for the *hazard.g* example, as shown in Figure 8).

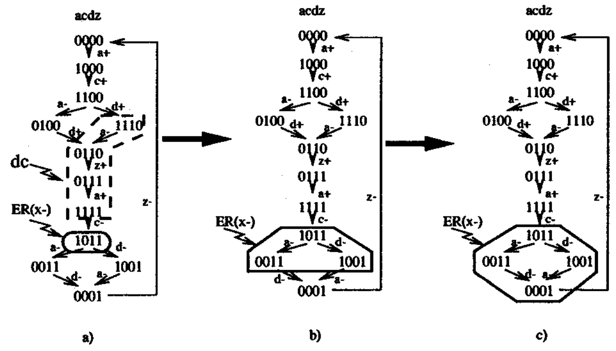

Figure 8: Derivation of $ER(x-)$ for a decomposition dc of a *hazard.g* example

Algorithm 4.1 Generation of ERs for a new signal, by example of $S^- = ER(x-)$

1. *Let $ER(x-) = IB(\overline{F})$*

2. *Find well-formed closure by recursive application of the following rule: if $s \in Pred(ER(x-)) \cap S^{\overline{F}}$, then let $ER(x-) = ER(x-) \cup s$.*

3. *Preserve (if required) the input-output interface by checking that no input signals can be delayed by x. For this do the following: for any input signal b: if $s \in after(b*, ER(x-))$, then let $ER(x-) = ER(x-) \cup s$.*

4. *Force SIP properties (make any intersection of state diamonds with $ER(x-)$ legal by inserting in $ER(x-)$ the corresponding states of the diamond). Goto Step 2.*

Calculation of $ER(x-)$ stops either if at some step $ER(x-)$ intersects with S^F (then there is no legal $ER(x-)$) or a fixed point is reached. Calculation of $ER(x+)$ is done similarly based on $IB(F)$.

Example *hazard.g* **continued.** In the example (see Figure 8,a) $ER(x-) = \{1011\}$ (step 1). It is well-formed (step 2). At step 3 we will find that state $0011 \in after(a-, ER(x-))$ and state $1001 \in after(d-, ER(x-))$. Therefore, $\{0011, 1001\}$ are included in $ER(x-)$ (see Figure 8,b). State diamond $\{1011, 0011, 1001, 0001\}$ illegally intersects $ER(x-)$ (step 4). To legalize this, the intersection state 0001 is included in $ER(x-)$ as shown in Figure8,c.

Figure 9 shows the results of $ER(x*)$ generation for the decomposition of $S_z = \overline{a}cd$ with divisors $\overline{a}d$, $\overline{a}c$ and dc, respectively. The choice $F = \overline{a}d$ is not valid (see Figure 9,a), because F intersects illegally with state diamond $\{1011, 0011, 1001, 0001\}$. This illegal intersection cannot be corrected by expanding $IB(F)$ without hitting states where $F = 0$. The divisors $\overline{a}c$ and dc are valid and the corresponding ERs of signal x are shown in Figure 9,b,c.

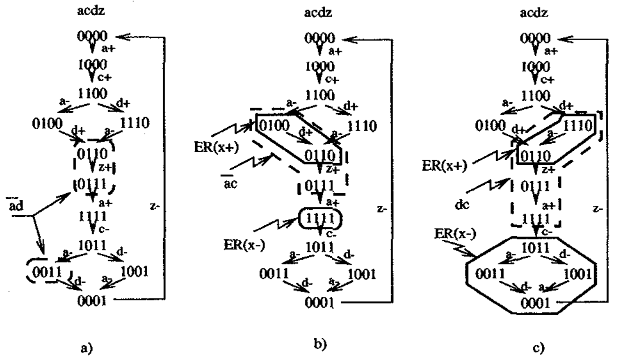

Figure 9: Three attempts to decompose $S_z = \overline{a}cd$ in *hazard.g* example

4.2 Progress Analysis

If $ER(x+)$ and $ER(x-)$ are derived, then there is a speed-independent implementation of the *SG* with a new signal x. However, to ensure progress in the technology mapping for the target cover function $C(a*) = F * G + R$, we would like to have the following implementation in the new circuit: $C(a*) = x * G + R$ (function F is substituted in this expression by one literal x)[4]. This is not always possible, since to preserve speed-independence, $C(a*)$ may require more fan-in signals. We will formulate progress conditions which will define when the implementation above is valid.

State images. The progress conditions are easily formulated in terms of the new *SG* A'. However, constructing the new *SG* is computationally hard and hence it is better to use the original *SG* A (cf. the approach in [4]). For this we need to compare the states of A and their images in A'. The insertion scheme (Figure 3) determines a binary relation (we call it an *image relation*) between the states of A and the states of A'. A state s' from *SG* A' is said to be *an image* of a state s from A if values of all signals, except x, are the same in s and in s'. Then, state s is called the *inverse image* of s'. The inverse image for any state from A' is unique. The opposite is not true. Each state $s \in ER(x*)$ from *SG* A has two images s', s'' in A' such that $s' \xrightarrow{x*} s''$. All other states in A have one image. The image relation is expanded to the sets of states. If S is a set of states in A', then its inverse image is denoted by S^{-1}. To avoid confusion, we will add subscript A or A' to address the objects in *SG*s A and A' if necessary.

Inverse images for excitation and quiescent regions. The validity of substituting a new signal x in a cover function $C(a*)$ is checked by considering the inverse images of $ER(a*)_{A'}$ and $QR(a*)_{A'}$. By construction, only states from $ER(a*)_A$ have images in which $a*$ is enabled, hence $ER(a*)_A$ is the inverse image of $ER(a*)_{A'}$. For quiescent regions the image relation is more complicated. Consider, for example, signal transition $a+$. For every state $s \in QR(a+)_A$ there is an image in which signal a is equal to 1, and therefore $QR(a+)_A \subseteq QR(a+)^{-1}_{A'}$. However, $QR(a+)^{-1}_{A'}$ can include additional states because some original signal transitions are delayed by x.

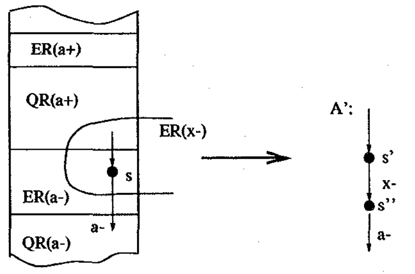

Figure 10: Inverse image for quiescent regions

This case is illustrated in Figure 10. In *SG* A state $s \in ER(a-)$. However, in one of its images, s', signal a is equal to 1 and is stable, and therefore $s' \in QR(a+)_{A'}$. Hence, state s is in the inverse image of $QR(a+)_{A'}$. The following procedure computes the inverse image for a quiescent region (by example of $QR(a_i+)^{-1}_{A'}$).

[4]Algorithm 4.1 does not modify the borders of S^F or $S^{\overline{F}}$, so the combinational solution $x = F$ is always valid. However the technique described in this section may also find a *sequential decomposition* with this combinational "seed".

Algorithm 4.2 Computing inverse image for quiescent regions

```
$QR(a_i+)^{-1}_{A'} = QR(a_i+)_A$;
**for each** $a_j-$ that succeeds $a_i+$ **do**
    **if** $s \in ER(x*) \cap ER(a_j-) \wedge after(a_j-, s) \cap ER(x*) = \emptyset$
        **then** $QR(a_i+)^{-1}_{A'} = QR(a_i+)^{-1}_{A'} \cup s$
**end for**
```

Before formulating progress conditions we present a useful property that captures conditions for signal x to have a constant value inside the excitation region of the original signal a in the *new SG* even if the excitation region for $x*$ in the *original SG* contains states from the $ER(a*)$ (x, as before, denotes the signal which is inserted for decomposition).

Property 4.3 *[8] Let* SG A' *be obtained from* SG A *by insertion of signal x. Let $a*$ be an event. Let $ER(x+)$ be a well-formed SIP closure of the input border for a block of a state partition for* SG A*, obtained with Algorithm 4.1, such that $ER(x+) \cap ER(a*) \neq \emptyset$. If the following two conditions are satisfied for* SG A
(1) $ER(x+) \cap after(a*, ER(a*)) = \emptyset$
(2) $ER(x-) \cap ER(a*) = \emptyset$,
then x is equal to 1 in any state of $ER(a*)_{A'}$.

A symmetrical property holds for $ER(x-)$. The next proposition states the progress condition by presenting conditions for preserving monotonous cover conditions for substituting function F with one literal x in the cover function $C(a*)$.

Proposition 4.1 *[7] Let $C_A(a*) = F * G + R$ be a monotonous cover of $ER(a*)$ in* SG A*. Let $ER(x+)$ and $ER(x-)$ be the $S^+$and S^-sets for inserting a signal x obtained by Algorithm 4.1. The function $C_{A'}(a*) = x*G+R$ satisfies the three conditions for the monotonous cover in the new* SG A'*, iff:*

1. Cover condition: $after(a*, (ER(a*) \cap F*G*\overline{R})) \cap ER(x+) = \emptyset$

2. One-hot condition: $\forall s : s \notin ER(a*) \cup QR(a*)^{-1}_{A'} \Rightarrow s \notin ER(x-) \cap G$

3. Monotonicity conditions:
(a) $\forall s : s \in (QR(a*) \cap F*G*\overline{R}) \Rightarrow s \notin ER(x+)$*, and*
(b) $\forall s : s \in QR(a*)^{-1}_{A'} \cap ER(x-) \cap G \Rightarrow Pred(s) \in G+R$

The proof is given in [8]. The conditions in the above proposition can be informally explained as follows.

Condition 1 ensures the cover condition for $C_{A'}(a*)$ in the new *SG* A', by detailing Property 4.3. Set $ER(a*) \cap F*G*\overline{R}$ contains those states of $ER(a*)$ in *SG* A that are covered by $F*G$, but not by R. Therefore, to satisfy the cover condition in *SG* A', the image of this set in A' must be covered by the function $x*G$. If $after(a*, (ER(a*) \cap F*G*\overline{R})) \cap ER(x+) \neq \emptyset$, then there is a transition $s_1 \overset{a*}{\rightarrow} s_2$ *internal* to $ER(x+)$ such that $F(s_1) = G(s_1) = 1$ and $R(s_1) = 0$. Hence, state s_1 has two images s_1' and s_1'' in A' such that $s_1' \overset{x+}{\rightarrow} s_1''$, which implies that signal x has value 0 in s_1' and value 1 in s_1''. Therefore, state $s_1' \in ER_{A'}(x+)$ is not covered by $C_{A'}(a*) = x*G+R$ since both $x*G$ and R have value 0 in s_1'. The cover condition is violated.

Condition 2 ensures the one-hot condition for $C_{A'}(a*)$ in the new *SG* A'. Let s be outside $ER(a*) \cup QR^{-1}(a*)$. If $s \in ER(x-) \cap G$ in *SG* A, then in the new *SG* A', function $x*G$ evaluates to 1 in the first image s' of s ($s' \overset{x-}{\rightarrow} s''$). Hence, for A', function $x*G$ is evaluates to 1 outside $ER_{A'}(a*) \cup QR_{A'}(a*)$, which violates the one-hot condition for the cover function $C_{A'}(a*) = x*G+R$.

Condition 3 ensures the monotonicity condition for $C_{A'}(a*)$ in *SG* A'. Condition 3(a) guarantees that $C_{A'}(a*)$ cannot make a non-monotonous transition of the type "1-0-1" along any path inside $ER_{A'}(a*) \cup QR_{A'}(a*)$. Set $QR(a*) \cap F*G*\overline{R}$ contains the states of $QR(a*)$ that are covered by $F*G$, but not by R, in *SG* A. Let some state s from this set belong to $ER(x+)$. Then there are two images for state s in *SG*, A': s' and s'' such that $s' \overset{x+}{\rightarrow} s''$. Function $x*G$ evaluates to 0 in s' and to 1 in s''. Neither image is covered by R. Moreover since states of $ER(a*)$ are covered by $C_{A'}(a*)$ the cover function $C_{A'}(a*)$ performs a non-monotonic transition 1-0-1 along a path within $ER_{A'}(a*) \cup QR_{A'}(a*)$ (this path starts in $ER(a*)$ and contains states s' and s'').
Condition 3(b) ensures that $C_{A'}(a*)$ cannot make a non-monotonous transition of the other type "0-1-0" along any path inside $ER_{A'}(a*) \cup QR_{A'}(a*)$. Assume that there is at least one state, s, such that $s \in QR(a*)^{-1}_{A'} \cap ER(x-) \cap G$ and let its predecessor, s_1, be covered neither by G nor by R. Then function $C_{A'}(a*)$ has value 0 in the image, s_1', of s_1 (if s_1 has two images, then $C_{A'}(a*)$ has value 0 in both). State s has two images in A' ($s' \overset{x-}{\rightarrow} s''$). Function $x*G$ evaluates to 1 in the first one, s', and to 0 in the second one, s''. Hence, function $C_{A'}(a*)$ performs a non-monotonous 0-1-0 transition along the path $s_1' \rightarrow s' \rightarrow s''$ in A'.

Example *hazard.g* **continued.** All the conditions of Proposition 4.1 are satisfied for $F = \overline{a}c$ and $F = dc$ and for both of them S_z can be safely decomposed into two AND gates.

4.3 Cost estimation

The progress condition (if satisfied) guarantees that the implementation of a target cover function $C(a*)$ will be simplified as a result of a decomposition. However, to accept a decomposition we need to check that it will not increase the complexity of logic for *other* events. We use a conservative estimate of logic complexity, in which trigger signals play a key role, in order to select candidates for decomposition.

All events (besides the target event $a*$) can be divided in 3 groups:

- *Events $x*$ of signal x*
  It can be shown, by analyzing the MC conditions that $x = F$ is a correct complete cover for a signal x.

- *Events for which $x*$ is not a trigger*
  The preconditions for these events are not modified by the insertion of x, and hence we *can* (in the

worst case) use the same implementation as before the decomposition. It is possible, though, that x can be used to further simplify the implementation of those signals as well, since the don't care set is increased.

- *Events for which $x*$ is a trigger*, denoted by $Tr(x)$. For estimating complexity of such events the following procedure is used.

Algorithm 4.3 Estimating complexity of signals for which x is a trigger

```
1. for each $b* \in Tr(x)$ do
  2. if $x*$ replaces trigger event $d*$ in $ER(b*)$ then
     /* property 4.4 */
    3. if $x$ substitutes $d$ in a cover function $C(b*)$ then
       /* proposition 4.2 */
      4. The complexity of $C(b*)$ is not increased
    5. else if $x$ can be added as one additional literal to $C(b*)$ then
       /* property 4.5 */
      6. The complexity of $C(b*)$ is increased moderately
    7. else
      8. Decomposition fails
    9. end if
  10. end if
11. end for
```

Further we consider the main steps of Algorithm 4.3.

Replacement of other trigger events by x (line 2 of Algorithm 4.3). Property 4.1 helps to find the set of events $Tr(x)$ for which signal x becomes a trigger. Conditions for replacing a trigger event by a new signal transition $x*$ are stated by the following property.

Property 4.4 *[8] An event $x*$ replaces $d*$ as a trigger event for $b*$ in* SG A' *iff in* SG A *the following conditions are satisfied:*
(1) $entry(d*, ER(b*)) \subset ER(x*)$
(2) $before(d*, ER(b*)) \cap ER(x*) = \emptyset$
(3) $after(b*, entry(d*, ER(b*))) \cap ER(x*) = \emptyset$

Example *hazard.g* **continued.** Let us consider a combinational decomposition of S_z using function $F = dc$. $ER(x+)$ satisfies all the conditions of Property 4.4 and hence $x+$ becomes a new trigger event for $z+$ instead of $d+$. On the other hand, for $ER(x-)$ for both events $a-$ and $d-$ condition 2 of Property 4.4 is violated. Therefore, events $a-$ and $d-$ are concurrent with $x-$ and none of them is replaced by the new trigger event $x-$. After inserting signal x event $z-$ will have *three* trigger events $x-, a-, d-$. For the decomposition based on function $F = \overline{a}c$, the new signal x replaces old trigger signals for both $z+$ and $z-$.

Validating substitution of signal x into a cover function other than $C(a*)$ (line 3 of Algorithm 4.3). If a trigger event $x*$ replaces another trigger event $d*$ for some $ER(b*)$, then the next step is to check that signal d can be replaced by signal x in the logic implementation of $C(b*)$. Assume that $C_A(b*) = d * M + N$. We want to check validity of substitution $C_{A'}(b*) = x * M + N$. Conditions for validity of such substitution are almost identical to those of Proposition 4.1.

Proposition 4.2 *Let $C(b*) = d * M + N$ be a monotonous cover of $ER(b*)$ in* SG A. *Let $\{S^+ = ER(x+), S^1, S^- = ER(x-), S^0\}$ be the I-partition for inserting signal x. The implementation $C_{A'}(b*) = x * M + N$ satisfies the three conditions for monotonous cover in the new* SG A' *iff:*

1. Cover condition: $(after(a*, (ER(a*) \cap d * M * \overline{N})) \cap ER(x+) = \emptyset) \wedge (ER(a*) \cap d * M * \overline{N} \cap ER(x-) = \emptyset)$

2. One-hot condition: $\forall s : s \notin ER(a*) \cup QR(a*)^{-1}_{A'} \Rightarrow s \notin (ER(x-) \cup S^1) \cap M$

3. Monotonicity conditions:
(a) $\forall s : s \in (QR(a*) \cap d * M * \overline{N} \Rightarrow s \notin ER(x+)$, *and*
(b) $\forall s : s \in QR(a*)^{-1}_{A'} \cap (ER(x-) \cup S^1) \cap M \Rightarrow Pred(s) \in M + N$

Let us clarify the difference between Propositions 4.1 and 4.2. In Proposition 4.1 signal x is the output of the gate implementing function F and is substituted into $C(a*) = F * G + R$ instead of F. In Proposition 4.2 x substitutes signal d, which is implemented by a gate different from the gate implementing x. Therefore, Conditions 1-3 have a more general form in Proposition 4.2. Indeed, to ensure the cover condition (according to Property 4.3) condition $ER(a*) \cap F * G * \overline{R} \cap ER(x-) = \emptyset$ is required. This condition is automatically satisfied if $x = F$ and x substitutes F in $C(a*)$, whereas it is not if x substitutes signal d. If signal x substitutes function F, x is equal to 1 in the same states as F with the exception of $ER(x*)$. Hence, in the *one-hot* and the *monotonicity* conditions, we should only consider states from $ER(x-)$. If x substitutes signal d, then states from S^1should be considered as well.

Note that Property 4.2 can also be used when signal x replaces several trigger signals $d_1, \ldots, d_k$. In this case the cover function for $b*$ can be represented as $C(b*) = d_1 * \ldots * d_k * M + N$. After substituting x a new cover function is $C(b*)_x = x * M + N$.

When the replacement fails (line 5 of Algorithm 4.3). In this case the complexity of a cover function for $ER(b*)$ can in general increase (unless the expanded don't care set induced by $x*$ implies further simplification of $C(b*)$). If the conditions of the following property are satisfied, then *no more than one literal* is added to the fan-in of $C(b*)$. We restrict our method with such a moderate increase in complexity only *to bound the search space.*

Property 4.5 *[7, 8] Let $C_A(b*)$ be a monotonous cover for event $b*$ in* SG A. *If in the* SG A' *obtained from A by inserting a new signal x the following conditions are satisfied:*

1. event $x+$ is a trigger for $b*$;

2. $ER(x+) \cap after(b*, ER(b*)) = \emptyset$ and

3. $C(b*) \cap ER(x-) = \emptyset$,

then the cover function $C_{A'}(b*) = C_A(b*) * x$ for event $b*$ in A' satisfies the monotonous cover conditions.

This property is used as a heuristic filter to select candidate divisors that are guaranteed not to increase excessively the complexity of the implementation of other signals.

Example *hazard.g* continued. For a decomposition with $F = dc$ (Figure 9,c) signal x becomes a new trigger for $z-$ without replacing any other trigger. Hence the cover for $z-$ will increase by one literal. A cover for $z+$ will decrease by one literal. This decomposition is not effective. If $F = \overline{a}c$ is used, then event $x-$ is inserted before $c-$ and replaces trigger event $a+$. Function for $c-$ will not increase in complexity. The result of decomposition using function $\overline{a}c$ is shown in Figure 6,b.

5 Sequential decomposition

5.1 Motivation

Combinational decomposition is limited, since signal insertion using bipartition $\{F, \overline{F}\}$ is based primarily on one of the two transitions of the gate's output (e.g., its rising transition). The other transition of the combinational gate is fully determined by the insertion place of the first transition. Moreover, if x substitutes F in cover function $C(a*) = F * G + R$, then in most cases, event $x+$ becomes a trigger to $a*$ and is acknowledged by $a*$ itself. However, $x-$ is often acknowledged by signals different from a, which may increase their complexity. Sequential decomposition, based on a new memory element, can improve the progress of mapping by allowing the opposite transition to play a more effective role, since the set and reset logic are inserted independently. In particular, two boolean functions can be decomposed at the same time with one new signal.

Assume that there are two functions $C(a*) = F*G+R$ and $C(b*) = P * Q + T$, which are not yet mapped in the library, and such that $F * P = 0$. Then, in the sequential decomposition, a new signal x is inserted in such a way that x will go to 1 when F is changing from 0 to 1, and go to 0 when P is changing from 0 to 1. Then, both rising and falling transitions of x can be used to simplify the cover functions: $x+$ to simplify $C(a*)$, and $x-$ to simplify $C(b*)$.

To illustrate that sequential decomposition can be more powerful than combinational decomposition, let us modify the *hazard.g* example, by declaring signal c to be an input (example *hazard-mod.g*). As before, we would like to map the three-literal function $S_z = \overline{a}dc$ into two-input gates. Since signal c is now an input, there are additional constraints for preserving the input/output interface. Indeed, we can no longer make the new event $x-$ trigger for input $c-$. Derivation of $ER(x+)$ and $ER(x-)$ for the example *hazard-mod.g* and the combinational decomposition based on $F = \overline{a}c$ (that succeeded in *hazard.g*) is shown in Figure 11,a. $ER(x-)$ for *hazard-mod.g* includes 5 states (instead of 1 for *hazard.g*, cf. Figure 9,a) and event $x-$ is no longer replacing any other trigger event of $z-$. Decomposition based on $F = \overline{a}c$ makes the cover function for event $z-$ even worse: 4 literals instead of 3 (hence it is not useful). Example *hazard-mod.g* cannot be mapped using only combinational decomposition. Further we will refer to *hazard-mod.g* to illustrate the steps of sequential decomposition.

5.2 State partitioning

Combinational decomposition using function F is based on bipartition of states into two blocks $S^F = \{s : F(s) = 1\}$ and $S^{\overline{F}} = \{s : F(s) = 0\} = \{s : \overline{F}(s) = 1\}$. This bipartition is transformed to an I-partition with four blocks for inserting a new signal x as described in Section 4.1.

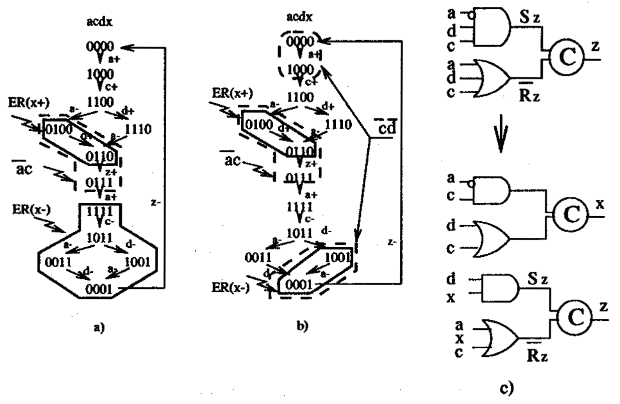

Figure 11: Decompositions of *hazard-mod.g* example: (a) combinational, (b) sequential, (c) logic for signal z

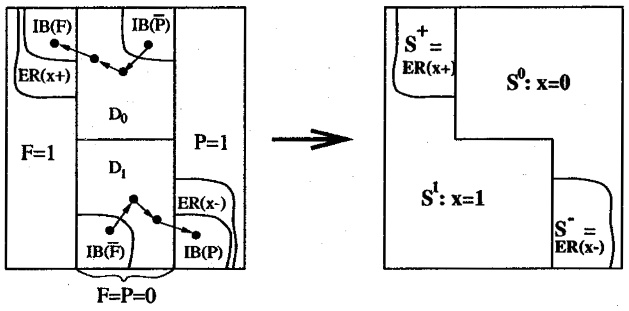

Figure 12: I-partition for sequential decomposition

Sequential decomposition using a pair of orthogonal functions $\{F, P\}$ [5] defines a partition of states into three blocks $\{S^F, S^P, S^{\overline{F}\,\overline{P}}\}$ (see Figure 12): $S^F = \{s : F(s) = 1\}$, $S^P = \{s : P(s) = 1\}$, and $S^{\overline{F}\,\overline{P}} = \{s : F(s) = P(s) = 0\}$. We make the following transformations of the three blocks when constructing an I-partition, $\{S^+, S^1, S^-, S^0\}$, based on the four input borders: $IB(F)$, $IB(P)$, $IB(\overline{F})$, and $IB(\overline{P})$ (see Figure 12)
.

Algorithm 5.1 Constructing I-partition for sequential decomposition

```
1. /* Construct $D_1$ and $D_0$ -- subsets of $S^{\overline{F}\,\overline{P}}$ */
   (a.i) $D_1$ = all states backward reachable from $Pred(IB(P))$ without hitting $S^F$;
   (a.ii) Include in $D_1$ all states forward reachable from $IB(\overline{F}) \cup D_1$ without hitting $IB(P)$ [6];
```

[5]As before, for an ease of presentation, we ignore in this paper the fact that the set and reset functions for C-elements are not required to be orthogonal.

[6]The reachability relation is reflexive ($s \Rightarrow s$) and hence states of $IB(\overline{F})$ can be included into D_1.

(b.i) D_0 = all states backward reachable from $Pred(IB(F))$ without hitting S^P;
(b.ii) Include in D_0 all states forward reachable from $IB(\overline{P}) \cup D_0$ without hitting $IB(F)$;

2. **If** $\neg(Succ(D1) \subseteq IB(P) \wedge Succ(D0) \subseteq IB(F))$ **then return** *failure;*

3. **If** *($D_1 \cap D_0 \neq \emptyset$)* **then return** *failure;*

4. *Construct $S^+ = ER(x+)$ as a well-formed SIP closure of $IB(F)$;*
   If $S^+ \cap (S^P \cup D_0) \neq \emptyset$ **then return** *failure;*

5. *Construct $S^- = ER(x-)$ as a well-formed SIP closure of $IB(P)$;*
   If $S^- \cap (S^F \cup D_1) \neq \emptyset$ **then return** *failure;*

6. *$S^1 = (S^F \cup D_1) - S^+$ and $S^0 = (S^P \cup D_0) - S^-$*
   / I-partition for $\{F, P\}$ is constructed */*

Algorithm 5.1 fails to construct an I-partition for $\{F, P\}$ if any one of steps 2-5 returns failure. Otherwise, the algorithm returns an I-partition $\{S^+ = ER(x+), S^1, S^- = ER(x-), S^0\}$.

Note that the set D_1 (and similarly D_0) is constructed in two steps 1(a.i) and 1(a.ii), by first applying the backward, and then the forward reachability. If *SG A* is cyclic (such that the initial state s_0 is reachable from any other state of a *SG*), then step 1(a.ii) can be omitted, since it does not produce any new states in D_1. However, if s_0 is not a cyclic state for *SG A* and $s_0 \in S^{\overline{F}\,\overline{P}}$, then both traversals are needed to identify which set, D_1 or D_0, state s_0 (and its successors) belongs to.

Algorithm 5.1 ensures consistency for the new signal x. Step 2 checks that any path in *SG A* starting from $IB(\overline{F})$ cannot reach $ER(x+)$ without crossing $ER(x-)$ (or symmetrically a path from $IB(\overline{P})$ cannot reach $ER(x-)$ without crossing $ER(x+)$). Step 3 checks that D_1 and D_0 have no states in common. These two checks guarantee that there are no cycles inside $D1 \cup D0$. Therefore, signal x can only perform consistent transitions in A': $1* \rightarrow 0 \rightarrow 0* \rightarrow 1 \rightarrow 1* \rightarrow \ldots$.

The following property, which proof can be found in [8], shows that Algorithm 5.1 is sound.

Property 5.1 *Let $I = \{S^+ = ER(x+), S^1, S^- = ER(x-), S^0\}$ be an I-partition obtained by Algorithm 5.1 for a pair of functions $\{F, P\}$ such that $F * G = 0$. The new* SG, *A', obtained from A by inserting signal x using I-partition I is consistent and speed-independent.*

Example *hazard-mod.g*. Let us consider the pair of functions $F = \overline{a}c$ and $P = \overline{d}\overline{c}$ (F is extracted from $S_z = \overline{a}cd$, while P is extracted from $R_z = \overline{a}\overline{d}\overline{c}$. The well-formed SIP sets $ER(x+)$ and $ER(x-)$ defined using $\{F, P\}$ are shown in Figure 11,b. The sequential decomposition based on $\{F, P\}$ is feasible because: 1) $F * P = 0$; 2) any cycle starting in $ER(x+)$ ($ER(x-)$) crosses $ER(x-)$ ($ER(x+)$) 3) $D_1 = \{1111, 1011, 0011\}$; $D_0 = \{1100, 1110\}$; $D_1 \cap D_0 = \emptyset$; and hence all conditions of Property 5.1 are satisfied.

5.3 Progress conditions

Sequential decomposition is aimed at mapping in the library two non-implementable cover functions $C(a*)$ an $C(b*)$. However, the decomposition can be useful if at least one of the functions is simplified. We can accept a sequential decomposition simplifying only one cover function (e.g., $C(a*) = F * G + R$) in two main cases:

1. Combinational decomposition using F for function $C(a*)$ failed because of the $x-$ event, e.g., the SIP conditions are violated for $ER(x-)$.

2. Combinational decomposition using F for function $C(a*)$ is valid, but it makes logic for some other (than $a*$) events more complex (e.g., due to the acknowledging event $x-$).

However, if none of the functions $C(a*)$ or $C(b*)$ is simplified by sequential decomposition, then it is rejected. Estimating progress for sequential decomposition is very similar to that for a combinational one.

- *For target events $a*$ and $b*$.* The validity of substituting signal x into $C(a*)$ and $C(b*)$ is checked by Proposition 4.2. If the substitution is not valid, the conditions of Property 4.5 are applied to implement $C_{A'}(a*)$ or $C_{A'}(b*)$ as $C_A(a*) * x$ or $C_A(b*) * \overline{x}$, correspondingly.

- *For events $x*$ of signal x.* If conditions of Property 5.1 are satisfied, then signal x can be implemented by a C-element with inputs F and $\overline{P}$.

- *Events for which x is not a trigger.* Similar to the combinational case, it can be shown that the complexity of these events cannot increase with the insertion of x.

- *Events for which x is a trigger.* We either check that x can substitute for some other trigger signals in these cover functions (see Proposition 4.2) or (if this check fails) that cover functions can be implemented with at most one extra literal x (see Property 4.5).

Example *hazard-mod.g* **continued.** For the target event $z+$, the sequential decomposition with $F = \overline{a}c$ and $P = \overline{c}\overline{d}$ satisfies the progress condition. It also does not disturb the implementability of event $z-$. Thus, the sequential decomposition is successful, while all combinational decompositions fail. The final implementation is shown in Figure 11,c.

6 Experimental results

The strategy for general logic decomposition presented above has been implemented and applied to a set of benchmarks. Results are shown in Table 1.

We have measured the complexity of each gate as the number of literals required to be implemented as a sum-of-product gate, either complemented or not. Thus a 2-input EXOR gate ($a\overline{b} + \overline{a}b$) is considered to be a 4-literal gate, whereas the function $ab + ac + db + dc$ is also considered a 4-literal gate ($\overline{\overline{a}\overline{d} + \overline{b}\overline{c}}$). This model is different from the one used in [4] where technology mapping was targeted at

Circuit	# gates with n literals						library (signals)/CPU			Siegel [15]	lits/latches ($i = 2$)		
	$n = 2$	3	4	5	6	7	$i = 2$	$i = 3$	$i = 4$	$i = 2$	non-SI	SI	SIS
alloc-outbound	4	2					1/7	–	–		16/3	15/4	22/0
chu133	2	2					2/2	–	–	yes	13/2	13/1	28/0
chu150	3	2					2/2	–	–	no	16/1	17/2	24/0
converta	5	2					1/2	–	–	no	21/3	15/3	57/0
dff	2	2					2/4	–	–	yes	14/2	14/2	26/0
ebergen	5	2					3/2	–	–	no	21/2	23/3	34/0
half	1	1					1/1	–	–	no	7/2	3/2	19/0
hazard	2	2					2/1	–	–	yes	14/2	14/2	28/0
master-read	4	4	1				8/274	1/39	–		37/7	37/9	93/0
mmu	4	2	1	1	1		n.i.	5/59	2/26				
mp-forward-pkt	2	2					2/3	–	–	yes	13/3	14/3	29/0
mr0	5	1	2	3	1	1	n.i.	5/130	4/91				
mr1	3	2	2	1	1		10/126	4/26	3/20		54/7	48/9	93/0
nak-pa	8	1					1/4.0	–	–	no	24/4	24/4	35/0
nowick	6	2					2/3	–	–	yes	21/3	18/1	20/0
pe-rcv-ifc	10	10	3	1			n.i.	3/450	1/203	no			
pe-send-ifc	10	7	7	1	1		n.i.	n.i.	n.i.				
ram-read-sbuf	4	3					2/8	–	–	yes	24/4	23/4	37/0
rcv-setup	1		1				2/2	1/1	–	yes	9/1	11/1	11/0
rpdft		1	3				5/10	2/3	–	yes	22/0	22/1	22/0
sbuf-ram-write	2	4					4/23	–	–	no	24/3	29/6	
sbuf-send-ctl	2		1				3/19	1/6	–		13/3	27/5	32/0
sbuf-send-pkt2	5	4					6/130	–	–	no	28/3	30/3	38/0
seq_mix	2	3		2			9/247	2/20	1/12		38/6	47/6	70/0
seq4	1	2		1			4/12	1/4	1/4		22/5	23/7	51/0
trimos-send	6		3				10/129	3/15	–		33/6	41/8	72/0
tsend-bm	8	6	4	2		1	n.i.	n.i.	n.i.				
vbe5b	3	1					1/1	–	–	no	13/2	13/2	35/0
vbe5c		1					1/1	–	–	no	7/3	6/3	16/0
vbe6a	4		4				8/31	4/11	–		38/6	19/7	82/0
vbe10b	2	5				1	10/76	3/20	1/6		43/7	33/7	95/0
wrdatab	5	7	1				7/93	2/11	–		52/5	54/6	90/0
Total											637/95	640/109	1243

Table 1: Experimental results

the implementation in FPGA 4-input lookup tables. Due to this fact, it is difficult to make direct comparison with the solutions from [4]. The difference between our approach and that of [4] can be easily shown by the example in Figure 13. For the STG of Figure 13 output signals z and y are implemented by 3-input AND gates. Our tool finds their decomposition into 2-input AND gates, in which both outputs z and y are used to acknowledge switchings of a new signal x. No valid decomposition (preserving speed-independence) exists when x is acknowledged by only one output (either y or z). The method from [4] looks for the decomposition within a single signal network and hence will fail to decompose 3-input AND gates.

The first set of columns in Table 1 indicates the complexity of the circuit before decomposition. The second set of columns reports the number of signals inserted for decomposition using gates with at most i literals ($i = 2, 3, 4$), and the CPU time required to find the solution (in seconds, for a Sparcstation 20). The number of inserted signals shows also the number of iterations in technology mapping -- the circuit is resynthesized every time a new signal is inserted. The next column summarizes the results presented by Siegel [15] about the implementability of the circuit with only 2-input gates. All realizations have been verified to be speed-independent.

From the 32 examples, only 5 were not implemented (*n.i.*) with 2-literal gates. Only one 5-input AND gate in *pe-send-ifc* and two 5-literal gates in *tsend-bm* were not decomposed when attempting to implement these circuits with 4-literal gates. We significantly improve over the results presented in [15], and only one circuit (*pe-rcv-if*) could not be realized with 2-literal gates from that benchmark suite.

The global-acknowledgment allows the method to effectively decompose complex gates with high fan-in (6 or 7 literals). This is shown by circuits like *mr1* and *vbe10b* that were implemented with 2-literal gates. Figure 14 illustrates this fact, depicting the circuit *mr1* before and after logic decomposition into 2-literal gates.

The effectiveness of sequential decomposition is illustrated in Figure 15. The insertion of a new latch was crucial to allow the decomposition of a gate that could not be decomposed by the approach presented in [15].

The final columns present a rough estimation of the cost for speed-independence-preserving logic decomposition. The cost is evaluated as the number of literals of the

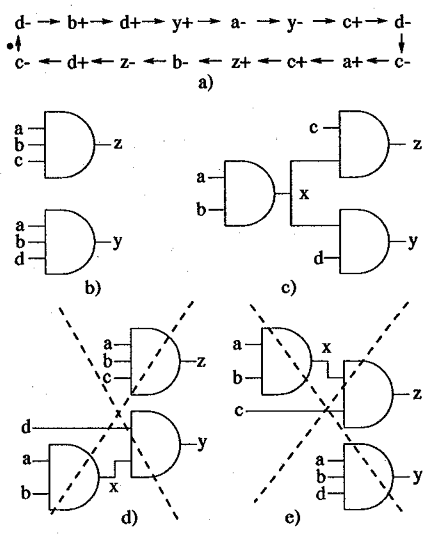

Figure 13: Example *abcd*: STG a) 3-AND gate implementation b) 2-AND gate implementation c) invalid "local" decompositions d,e).

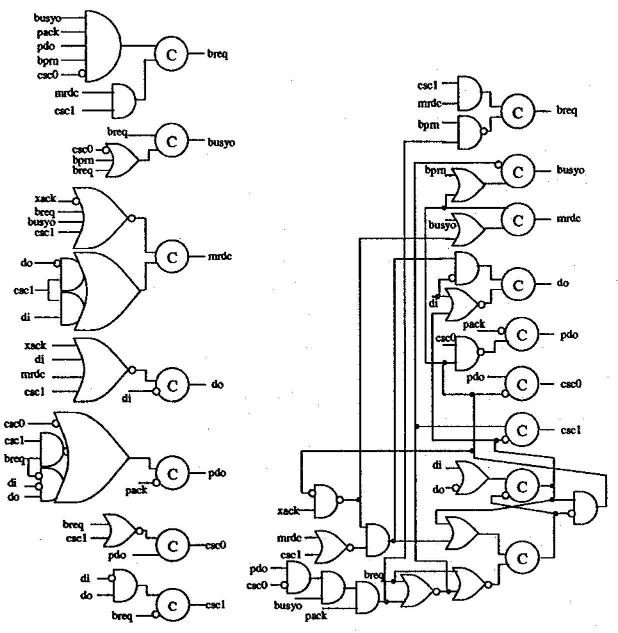

Figure 14: *mr1* before and after logic decomposition into 2-literal gates.

combinational gates and the number of C elements of the circuit. The column "*non-SI*" reports the cost of decomposing the original implementation of the circuit into 2-literal gates without preserving speed-independence (*tech_decomp -a 2* command in SIS). The column "*SI*" reports the cost of the decomposition preserving speed-independence. In some cases, such as *vbe6a*, the number of literals is reduced because the decomposition strategy allows sharing logic among different covers. In most cases extra cost is added to preserve speed-independence. However, if we consider that the area of a C element is roughly equivalent to a 3-input AND gate, we can conclude that the area cost of preserving speed-independence is not higher than 5%.

The last column shows, for the sake of comparison, the cost of performing technology mapping against a 2-input library (which is roughly the same as a 2-literal library) using the *bounded wire delay model*, after delay padding ([10]). If we consider the cost of a C element to be 3 literals, the total cost of the speed-independent implementations in the 2-literal library is $640 + 109 \times 3 = 967$ literals, which is considerably smaller (and probably faster, because there is no need to add delay buffers).

7 Conclusions and future work

In this paper we have shown a solution to the problem of multi-level logic synthesis and technology mapping for asynchronous speed-independent circuits. The method is based on both combinational and sequential decomposition, for each of which we apply a two-step approach.

The first step (Section 3) chooses a *candidate* for decomposition: algebraic kernels, non-cube-free sub-SOPs, sub-cubes etc. Different versions are evaluated and the "best" is taken -- say, it corresponds to the new signal x. Combinational decomposition for synchronous circuits stops here. In the case of sequential decomposition two candidates are considered simultaneously.

The second step (Sections 4 and 5) performs *actual* decomposition -- it attempts to find an optimized speed-independent implementation based on the candidate obtained at step (1). This is based on partitioning the state space into four sets, in which signal x is stable and is changing, while ensuring the speed-independence of the expanded specification (a necessary condition for speed-independent implementability). A new implementation is then derived for *each signal*, thus achieving global optimization and acknowledgement. The complexity arguments in Section 4.3 show that there is a good chance that x will get exactly the same function which was extracted at step (1). However, there is a chance also that this function will be smaller (thanks to boolean decomposition). Multiple acknowledgments for x appear automatically at this function generation step. Functions for signals which were not decomposed at step (1) may also change. Whenever a combinational decomposition fails to simplify the overall complexity (due to the lack of control in the insertion of the opposite transition $x-$), the procedure applies a sequential decomposition (where $x-$ is used to simplify one more cover). As a result, the actual function for x may correspond to a very general sequential decomposition. Moreover this is not a local, but "global" decomposition since other signals may change as well.

The method is implemented in the tool **petrify**. The results shown in the last section, to the best of our knowledge, show that the method appears to be the most effective and efficient amongst those available to date for the standard set of asynchronous benchmarks For example, it is for the first time that such examples as vbe10 and wrdatab have been decomposed into two-input AND gates by a software tool.

We are currently working at improving the method to make it complete (i.e. answering the key question of what is

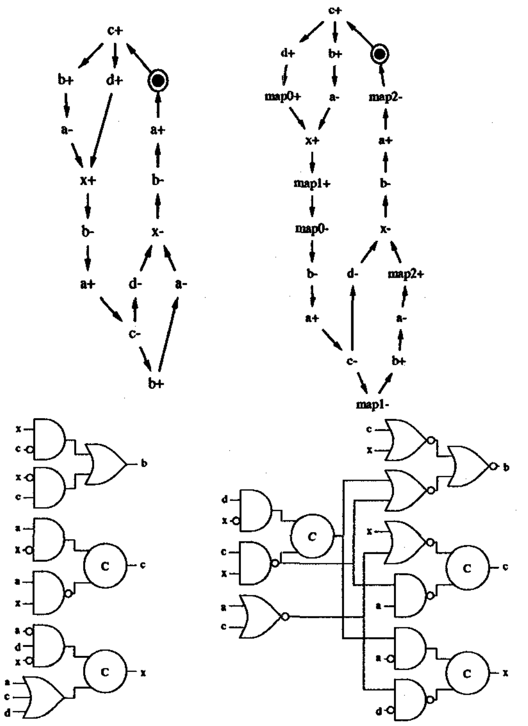

Figure 15: Example *ebergen* before and after logic decomposition into 2-literal gates.

the largest class of State Graphs that can be implemented in a given library) and at extending the basic implementation architecture to other types of sequential elements, such as S/R flip-flops or D latches.

References

[1] P. A. Beerel and T. H-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[2] Peter A. Beerel and Teresa H.-Y. Meng. Logic transformations and observability don't cares in speed-independent circuits. In *Proceedings of TAU 1993*, September 1993.

[3] R.K. Brayton, G.D. Hatchel, and A.L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of IEEE*, 78(2):264--300, February 1990.

[4] S. Burns. General conditions for the decomposition of state holding elements. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems, Aizu, Japan*, March 1996.

[5] S. Burns and A. Martin. A synthesis method for self-timed VLSI circuits. In *Proceedings of the International Conference on Computer Design*, 1987.

[6] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Complete state encoding based on the theory of regions. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems, Aizu, Japan*, March 1996.

[7] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Technology mapping of speed-independent circuits based on combinational decomposition and resynthesis. In *Proc. of European Design and Test Conference*, Paris(France), March 1997.

[8] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Technology mapping for speed-independent circuits. Technical Report TR 96-2-005, Aizu University, 1996.

[9] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits. In *Proceedings of the Design Automation Conference*, 1994.

[10] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits.* Kluwer Academic Publishers, 1993.

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

[12] Chris J. Myers, Peter A. Beerel, and Teresa H.-Y. Meng. Technology mapping of timed circuits. In *Asynchronous Design Methodologies*, pages 138--147. IEEE Computer Society Press, May 1995.

[13] Enric Pastor, Jordi Cortadella, Alex Kondratyev, and Oriol Roig. Structural methods for the synthesis of speed-independent circuits. In *Proc. of European Design and Test Conference*, pages 340 -- 347, Paris(France), March 1996.

[14] M. Sawasaki, C. Ykman-Couvreur, and B. Lin. Externally hazard-free implementations of asynchronous circuits. In *Proceedings of the Design Automation Conference*, June 1995.

[15] P. Siegel and G. De Micheli. Decomposition methods for library binding of speed-independent asynchronous designs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 558--565, November 1994.

[16] P. Siegel, G. De Micheli, and D. Dill. Automatic technology mapping for generalized fundamental mode asynchronous designs. In *Proceedings of the Design Automation Conference*, June 1993.

[17] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A generalized state assignment theory for transformations on Signal Transition Graphs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 112--117, November 1992.

[18] V. I. Varshavsky, M. A. Kishinevsky, V. B. Marakhovsky, V. A. Peschansky, L. Y. Rosenblum, A. R. Taubin, and B. S. Tzirlin. *Self-timed Control of Concurrent Processes.* Kluwer Academic Publisher, 1990. (Russian edition: 1986).