

# Synthesis of Asynchronous Circuits for Stuck-at and Robust Path Delay Fault Testability\*

Steven M. Nowick  
Dept. of Comp. Science  
Columbia University  
New York, NY 10027

Niraj K. Jha  
Dept. of Electrical Engg.  
Princeton University  
Princeton, NJ 08544

Fu-Chiung Cheng  
Dept. of Comp. Science  
Columbia University  
New York, NY 10027

## Abstract

*In this paper, we present methods for synthesizing multi-level asynchronous circuits to be both hazard-free and completely testable. Making an asynchronous two-level circuit hazard-free usually requires the introduction of either redundant or non-prime cubes, or both. This adversely affects its testability. However, using extra inputs, which is seldom necessary, and a synthesis for testability method, we convert the two-level circuit into a multi-level circuit which is completely testable. To avoid the addition of extra inputs as much as possible, we introduce new exact minimization algorithms for hazard-free two-level logic where we first minimize the number of redundant cubes and then minimize the number of non-prime cubes. We target both the stuck-at and robust path delay fault models using similar methods. However, the area overhead for the latter may be slightly higher than for the former.*

---

\*This work was supported in part by AT&T under a Special-Purpose Grant Award, by NSF under Grant no. MIP-9308810, and by an Alfred P. Sloan Research Fellowship.

# 1 Introduction

Achieving complete testability of asynchronous circuits has long been recognized to be a difficult problem since these circuits must be hazard-free [15]. Hazard-free synthesis methods frequently introduce redundant or non-prime product terms, resulting in circuits which are not fully testable. Thus, ensuring hazard-free behavior and at the same time achieving complete testability seem to be contradictory requirements. However, our aim in this paper is to show that hazard-free completely testable asynchronous multi-level circuits can be easily synthesized, in some rare cases requiring some extra control inputs.

In order to ensure high reliability of a circuit, one must test both its logical and temporal behavior for correctness. Physical defects may increase the propagation delays along different paths, giving rise to *delay faults* [19]. Delay faults can be categorized according to two models: *gate delay faults* and *path delay faults*. The gate delay fault model models excessive delay limited to just one gate, whereas the path delay fault model models excessive delays along a whole path from an input to an output. Therefore, the path delay fault model is more comprehensive; however, it may require more time for test generation because the number of paths is usually much larger than the number of gates.

Delay faults are generally tested by two-pattern tests. For path delay faults, these tests launch a  $0 \rightarrow 1$  or a  $1 \rightarrow 0$  transition at the input of the path to see if the desired transition reaches the output of the path within the specified time. A two-pattern test is called *robust* if arbitrary delays elsewhere in the circuit cannot invalidate it [19]. A robust test can be further categorized into a *hazard-free* or *non-hazard-free* test. For a hazard-free robust test, no hazards can occur on the tested path irrespective of the delay values elsewhere in the circuit. This is the most stringent fault model. Hazard-free robust path delay fault testability of a circuit also implies testability under other fault models, such as stuck-open [17]. Since it is known that robust testability of general circuits is usually quite low [19], many synthesis for testability methods for this fault model have been presented [17], [20]-[26]. However, these methods are not geared towards hazard-free implementations that are required for asynchronous circuits.

Several efforts have been made to attack the asynchronous testing problem [1]-[8]. For example, for a class of asynchronous circuits called *speed-independent*, the presence of some stuck-at faults can stop the circuit, thereby providing a degree of self-checking behavior [1, 2]. However, not all stuck-at faults stop the circuit. Other testability methods have been introduced to handle particular design styles such as *micropipelines* [5] and *Tangram*-based designs from Phillips [6]. A comprehensive synthesis for

testability method has been proposed by Keutzer *et al.* [4]. This work targets the hazard-free robust path delay fault model, which is one of the fault models we target too. The rationale for using this fault model is that since asynchronous interface circuits usually have strict delay requirements, it is desirable to detect any abnormality in their temporal behavior. Another interesting method has also been developed, where independent control of both phases of each variable is assumed [7].

The aim of this paper is to synthesize hazard-free asynchronous circuits which are also completely testable under the stuck-at or the robust path delay fault models. Both fault models are treated similarly; however, the latter model may require slightly greater area overhead. As in [4], we assume that full scan [8, 9] is available for converting the asynchronous circuit memory elements, such as Set-Reset or C-element, into scanned memory elements in order to make these elements controllable and observable. For delay fault testing, enhanced scan [10], which allows the application of two bits in sequence to the present state lines, is assumed.

Our method differs from [4] in several ways. First, they start with a two-level circuit which is guaranteed to be prime but may be redundant, whereas in our work the two-level circuit may be both redundant and non-prime. Second, our method includes two-level synthesis algorithms which exactly minimize redundancy and non-prime implicants, thereby enhancing two-level testability, while their method does not. Finally, their multi-level synthesis approach is an adaptation of a previous synthesis for testability method based on Shannon's decomposition [17], whereas we use an alternative method. They also present a heuristic method based on algebraic factorization [31], which can be shown to be a special case of our approach. Our own method is adapted from a previous synthesis for delay fault testability method for combinational circuits [23], which assumed prime and irredundant two-level circuits as a starting point, and thus was not geared towards asynchronous circuits.

Our synthesis method has three steps:

- (i) We synthesize a two-level circuit which is hazard-free, using a new algorithm which first minimizes the number of redundant products and then minimizes the number of non-prime products. The motivation is to reduce the need for extra control inputs (a different approach for using extra inputs for robust testability has been given in [24]).
- (ii) The hazard-free two-level circuit is converted into a multi-level circuit which is *completely testable* for the given fault model, yet maintains the hazard-free property of the original circuit.
- (iii) The multi-level circuit is then further optimized, using multi-level testability- and hazard-preserving transformations, to obtain the final circuit.

## 2 Synthesis of Optimally-Testable Two-Level Hazard-Free Logic

In this section, we address the problem of synthesis for testability of hazard-free two-level logic. Such logic, in general, has two features which pose problems for testing: the presence of (a) non-prime implicants, and (b) redundancy. We first present background on combinational hazards and review an existing algorithm for hazard-free two-level logic minimization [11]. We then address the synthesis for testability problem by extending this algorithm in two ways. Our first algorithm finds a hazard-free solution with exactly minimum number of non-primes; our second algorithm finds a hazard-free solution having exactly minimum redundancy. The two algorithms are then combined into a single algorithm which finds an *optimally-testable* hazard-free two-level solution using a three-level cost function: (i) minimum redundancy (primary cost), (ii) minimum number of non-primes (secondary cost), and (iii) minimum number of implicants (tertiary cost).

### 2.1 Combinational Hazards

For the following discussion, a combinational circuit model is assumed where gates and wires may have arbitrary finite delays. Since we are concerned with the dynamic behavior of a combinational circuit as its inputs change value, we need to formalize the notion of a “multiple-input change”.

A *transition cube* [15, 12, 11] is a cube with a *start point* and an *end point*. Given input states  $A$  and  $B$ , the transition cube  $[A, B]$  has start (end) point  $A$  ( $B$ ) and contains all minterms that can be reached during a transition from  $A$  to  $B$ . The cube describes a *multiple-input change* or *input transition* from  $A$  to  $B$ . Inputs are assumed to change monotonically (*i.e.*, at most once) in any order and at any time. Once a multiple-input change occurs, no further inputs may change until the circuit has stabilized.

A function  $f$  which does not change monotonically during an input transition is said to have a **function hazard** [15, 12]. For example, in Figure 1(a), the given function  $f$  has a *static function hazard* in the input transition from minterm 1110 to 1011, since  $f$  has the same value (0) at the start and end points, but an intermediate minterm may be reached with a different value ( $f(1111) = 1$ ). Function  $f$  has a *dynamic function hazard* in the the input transition from 1010 to 0111, since  $f$  is 0 at the start point and 1 at the end point, and there is a path from the start point to end point where  $f$  changes more than once:  $f(1010) = 0$ ;  $f(0010) = 1$ ;  $f(0011) = 0$ ;  $f(0111) = 1$ .

If an input transition has a function hazard, *no* implementation of the function is guaranteed to avoid glitches during the transition (assuming our circuit model of arbitrary gate and wire delays) [15].

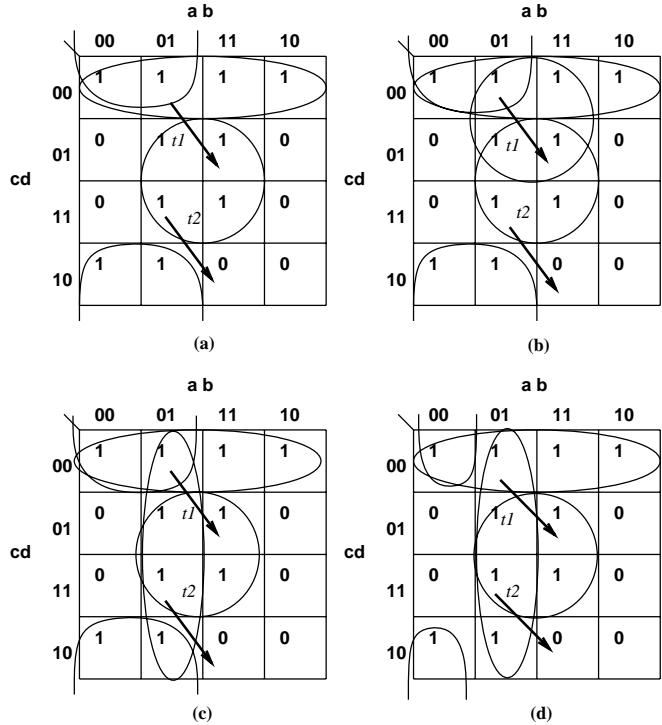


Figure 1: Hazard example

Henceforth, we consider only input transitions which are *function-hazard-free*. Given a function-hazard-free transition, a circuit implementation of  $f$  may still glitch, due to delays in the actual gates and wires. In this case, the circuit is said to have a **logic hazard** for the input transition. The hazard is *static* for an input transition from  $A$  to  $B$  if  $f(A) = f(B)$  and *dynamic* if  $f(A) \neq f(B)$ .

## 2.2 Conditions for a Hazard-Free Transition

We now describe conditions to avoid logic hazards in a sum-of-products implementation (for further details, see [12, 11]). These conditions are best illustrated by examples.

**Example 2.1.** Again, consider the example in Figure 1(a). A minimum-cost cover is shown containing three products. The corresponding sum-of-products implementation has a *static logic hazard* for input transition  $t1$  from  $abcd = 0100$  to  $abcd = 1101$ . Initially, two products are high:  $\bar{c}\bar{d}$  and  $\bar{a}\bar{d}$ ; during the transition,  $\bar{c}\bar{d}$  and  $\bar{a}\bar{d}$  go low and  $bd$  goes high. As a result, assuming arbitrary gate and wire delays,  $\bar{c}\bar{d}$  and  $\bar{a}\bar{d}$  may go low *before*  $bd$  goes high, and the OR gate output may glitch. The cover in Figure 1(b) solves the problem. A fourth cube is included,  $b\bar{c}$ , which *remains at 1* throughout the entire input transition. Therefore, the OR gate output remains at 1, and the transition is logic-hazard-free.  $\square$

In this example, a  $1 \rightarrow 1$  static logic hazard is avoided by insuring that the entire transition cube

$[0100, 1101]$  is *completely contained* in some product of the cover. This transition cube  $[0100, 1101]$  is called a **required cube** [11], since it is required that this cube be contained in some product in the cover. Note that, in this example, a *redundant product*  $b\bar{c}$  was added to the cover in Figure 1(b).

**Example 2.2.** Next, consider a *dynamic transition*  $t2$  in Figure 1(a) from  $abcd = 0111$  to  $abcd = 1110$ . A necessary condition for insuring that  $t2$  is hazard-free is to insure that each  $1 \rightarrow 1$  *static sub-transition* of  $t2$  is also hazard-free. For example, if *only* input  $d$  goes low, the circuit must still be hazard-free. However, for this sub-transition the circuit is not hazard-free: the transition cube  $[0111, 0110]$  is not contained in any product of the cover, and, therefore, the sub-transition has a static logic hazard. The cover of Figure 1(c) solves this problem: for each  $1 \rightarrow 1$  sub-transition within  $t2$ , the corresponding transition cube is contained in a product of the cover:  $[0111, 0110] \subseteq \bar{a}b$  and  $[0111, 1111] \subseteq bd$ . Transition cubes  $[0111, 0110]$  and  $[0111, 1111]$  are *required cubes* of the dynamic transition  $t2$ .

While the cover of Figure 1(c) insures that each static sub-transition of  $t2$  is hazard-free, it still does not guarantee that dynamic transition  $t2$  itself is hazard-free. In fact,  $t2$  has a hazard. Initially  $\bar{a}\bar{d}$  is low; when  $d$  goes low,  $\bar{a}\bar{d}$  goes high; finally, when  $a$  goes high,  $\bar{a}\bar{d}$  goes low. Therefore,  $\bar{a}\bar{d}$  may glitch during the transition. Assuming arbitrary delays on gates and wires, this glitch may propagate to the OR gate output, and the result is a dynamic logic hazard. The problem can be seen in the Karnaugh map:  $\bar{a}\bar{d}$  intersects transition  $t2$  in the middle, but *not* at its *start point*  $0111$ . Such an intersection is called *illegal* [11]; as a result,  $\bar{a}\bar{d}$  may become enabled, then disabled, during the transition. The cover of Figure 1(d) solves the problem. First, each static  $1 \rightarrow 1$  sub-transition is hazard-free (since  $[0111, 0110] \subseteq \bar{a}b$  and  $[0111, 1111] \subseteq bd$ ). Second, no product illegally intersects dynamic transition  $t2$ . This last condition insures no dynamic hazards.  $\square$

In this example, note that to avoid an illegal intersection, product  $\bar{a}\bar{d}$  was reduced to  $\bar{a}\bar{b}\bar{d}$ , which is in fact *non-prime*.

In summary, Examples 2.1 and 2.2 illustrate the conditions which must be satisfied to insure hazard-free two-level logic for  $1 \rightarrow 1$  and  $1 \rightarrow 0$  transitions. For the  $1 \rightarrow 1$  case, the entire transition cube is called a *required cube*, and must be completely contained in some product. For the  $1 \rightarrow 0$  case, first, each  $1 \rightarrow 1$  sub-transition must be hazard-free, so the corresponding required cubes must each be contained in some product. Second, no product in the cover may *illegally intersect* the  $1 \rightarrow 0$  transition, otherwise a dynamic hazard will result. Satisfying these conditions may require use of redundant and non-prime implicants.

There are two remaining transitions. A  $0 \rightarrow 1$  transition may be regarded as a  $1 \rightarrow 0$  transition

in reverse, and the same conditions apply. Finally, for a  $0 \rightarrow 0$  transition, there are no additional constraints. That is, every (function hazard-free)  $0 \rightarrow 0$  transition is free of logic hazards in any sum-of-products implementation [15, 12, 11].

### 2.3 Hazard-Free Covers

A *hazard-free cover* is a cover of a function which is hazard-free for a *set* of specified input transitions. The following theorem formulates the hazard-free covering problem:

**Theorem 2.1.** [11] A set of implicants  $C$  is a hazard-free cover for function  $f$  with respect to a specified set of input transitions if and only if: (a) each *required cube* of  $f$  is contained in some implicant in  $C$ , and (b) no implicant of  $C$  *illegally intersects* a specified dynamic transition.

An implicant which does not illegally intersect any dynamic transition is called a *dynamic-hazard-free implicant* (or *dhf-implicant*). Only dhf-implicants may appear in a hazard-free cover. A *dhf-prime implicant* is a dhf-implicant contained in no other dhf-implicant. An *essential dhf-prime implicant* is a dhf-prime implicant which contains a required cube contained in no other dhf-prime implicant.

### 2.4 Hazard-Free Two-Level Logic Minimization

Using Theorem 2.1, the *two-level hazard-free logic minimization problem* is to find a minimum-cost cover of a function using only dhf-prime implicants where every required cube is covered. This problem is a variant of the classic two-level minimization problem, where each ON-set minterm of a function must be covered by a prime implicant [16, 27]. An exact hazard-free two-level minimizer has been developed [11], based on this theorem, using a constrained version of the Quine-McCluskey algorithm. There are three steps:

1. Generate the *dhf-prime implicants* of a function;
2. Construct a *dhf-prime implicant table*; and
3. Find a minimum cover of this table.

The dhf-prime implicant table sets up the hazard-free covering problem: rows of the table are dhf-prime implicants and columns are required cubes. A solution consists of a set of rows covering all the columns. The core of the algorithm in Step 3 is the following loop for table reduction:

```

Algorithm reduce-tab (tab):
  essen-implicants = {};
  repeat
    do-row-dominance (tab);

```

```

do-column-dominance (tab);
essen-implicants = essen-implicants  $\cup$  get-essential-implicants (tab);
until no change;
return (essen-implicants).

```

Once table reduction is completed, the reduced table, if non-empty, describes a *cyclic covering problem* [27]. It can be solved using brute-force [16] or optimized search techniques [27].

## 2.5 Algorithms for Hazard-Free Synthesis for Testability

We now consider changes to the existing hazard-free two-level minimization algorithm to synthesize optimally-testable logic. For this paper, we focus on modifications to algorithm *reduce-tab*.<sup>1</sup>

Steps “do-column-dominance” and “get-essential-implicants” of *reduce-tab* are unchanged, since they perform structural simplifications on the covering table. Therefore, only step “do-row-dominance” needs to be changed. In classic combinational logic synthesis, given rows  $i$  and  $j$  of an implicant table, row  $i$  is said to *row-dominate*  $j$  if  $i$  covers all the columns covered by  $j$ ; that is:  $cols(j) \subseteq cols(i)$ .

### 2.5.1 Minimizing Non-Primes

The first problem is to find a hazard-free cover with an exactly minimum number of non-prime implicants. To do so, a small modification is necessary to “row-dominance”. If row  $i$  row-dominates  $j$ , where  $i$  is *non-prime* and  $j$  is prime, it is possible that a solution using  $j$  *instead* of  $i$  may have fewer non-primes and, therefore, be preferable. However, in all other cases, it is safe for  $i$  to dominate  $j$ : both prime, both non-prime, or  $i$  prime and  $j$  not. This modified row-dominance algorithm is *non-prime non-increasing (npni)*, since it guarantees that a prime is never rejected in favor of a non-prime.

```

Algorithm npni-row-dominate (i j):
  if (cols(j)  $\not\subseteq$  cols(i))
    return (false);
  else if (i is non-prime and j is prime)
    return (false);

```

---

<sup>1</sup>If *reduce-tab* yields a non-empty reduced table, it can be solved using Petrick’s method [16]. Petrick’s method is modified in a straightforward way to select solutions based on the desired cost function: minimizing non-primes or minimizing redundancy.

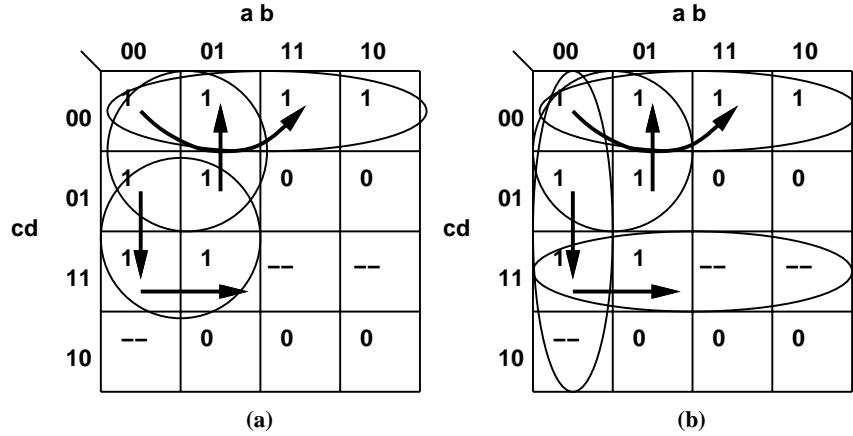


Figure 2: Minimizing redundancy in hazard-free logic

*else return (true).*

### 2.5.2 Minimizing Redundancy

The second problem is to find a hazard-free cover with an exactly minimum number of redundant implicants. This problem is more subtle than the problem of non-primes, since a product is itself either prime or non-prime while the redundancy of a product depends on the cover in which it is embedded.

For testability purposes, a product  $p$  in a cover  $C$  is *redundant* if and only if  $p$  is covered by  $C - \{p\}$ . In contrast, logic minimization algorithms [27] often take into account the “don’t-care set”,  $D$ , and define  $p$  to be redundant if and only if  $p$  is covered by  $C \cup D - \{p\}$ . This latter definition, however, is less relevant to testing.

The motivation in redundancy minimization is to accept covers with possibly additional products if redundancy can be reduced. As an example, consider the Karnaugh maps in Figure 2 for a function  $f$  with the specified input transitions, as shown. Figures 2(a) and (b) illustrate two hazard-free covers for  $f$  for the set of given transitions. The cover in Figure 2(a) has minimum cardinality of all hazard-free covers, with three products. However,  $\bar{a}\bar{c}$  is redundant. The cover in Figure 2(b) has four products, but no redundant products. Therefore, there are cases where redundancy can be decreased by allowing additional products.

There are two ways in which row-dominance by a row  $i$  may be unsafe. First,  $i$  itself may be redundant in a resulting cover, where the dominated row,  $j$ , is not. Second,  $i$  may introduce redundancies

into other products where  $j$  does not.

Algorithm *rni-row-dominate-unopt*, shown below, is an algorithm for *redundancy non-increasing (rni)* row-dominance. It requires that  $cols(j) \subseteq cols(i)$  and that  $i$  itself be irredundant in the current cover  $C$  described by the implicant table. It then checks whether region  $i \# j$  (i.e.,  $i$ , with the minterms of  $i \cap j$  removed) intersects any product  $p$  which is currently redundant in  $C_{new}$ . If so, it returns *false*, on the conservative assumption that  $i$  might contribute to the redundancy of  $p$  as the table is solved. Otherwise, the algorithm returns *true*.

*Algorithm rni-row-dominate-unopt* (i j):

```

C = set of rows in table;
Cnew = C - {j};
if (cols(j) ⊈ cols(i))
    return (false);
else if (i is redundant in Cnew)
    return (false);
else {
    new-region = i # j;
    for each cube p ∈ Cnew where p ≠ i
        if (p ∩ new-region ≠ ∅) and
            (p is redundant in Cnew)
            return (false);
    return (true). }

```

Algorithm *rni-row-dominate-opt* takes a similar but less conservative approach. Suppose  $i$  intersects some essential dhf-prime implicant  $e$ , where  $e$  is irredundant.<sup>2</sup> In this case, the region of intersection,  $i \cap e$ , is “safe” because it is guaranteed to be covered by  $e$  in the final cover. Therefore,  $i$  cannot introduce avoidable redundancies in the cubes which intersect this region. Hence, this region can be ignored.

---

<sup>2</sup>In hazard-free minimization, a product may be essential with respect to covering of required cubes, and yet still be redundant.

*Algorithm rni-row-dominate-opt* (i j):

$C$  = set of rows in table;

$C_{new} = C - \{j\}$ ;

safe-set = set of irredundant essential  
dhf-prime implicants in  $C_{new}$ ;

*if* ( $cols(j) \not\subseteq cols(i)$ )  
    *return* (false);

*else if* ( $i$  is redundant in  $C_{new}$ )  
    *return* (false);

*else* {  
    reduced-i-region =  $i \# (\text{safe-set} \cup \{j\})$ ;

*for* each cube  $p \in C_{new}$  where  $p \neq i$   
        *if* ( $p \cap \text{reduced-i-region} \neq \phi$ ) and  
            ( $p$  is redundant in  $C_{new}$ )  
            *return* (false);

*return* (true). }

We have implemented the new synthesis for testability algorithm by combining algorithms “npni-row-dominate” and “rni-row-dominate-opt” into a single algorithm, which first finds minimum-redundancy solutions, and of these picks those with fewest non-primes. Of the resulting solutions, it then picks one with fewest products.

### 3 Synthesis of Completely Testable Multi-Level Hazard-Free Logic

In the previous section, we presented a method which produces a hazard-free two-level logic, where we try to avoid redundant product terms as much as possible, while allowing non-prime product terms, whenever necessary. This two-level circuit synthesis method may produce any of the four possibilities where the circuit is redundant or irredundant and prime or non-prime. It would be easier to understand the method for obtaining a fully testable multi-level logic from this two-level logic, if we discuss the four cases separately, and try to unify the method during this discussion. Our approach would be to first convert the not-fully-testable two-level logic to a testable three or four-level logic, whenever possible, or a testable two-level logic with the help of extra inputs. Then we will use multi-level

testability/hazard-preserving transformations, such as algebraic factorization [31], to obtain the final multi-level circuit. Before we proceed, we should mention some previous general results which our method takes advantage of, as follows.

- An algebraic factorization method based on single and double cube divisors and their complement preserves single stuck-at fault testability [28]. This result is itself an extension of previous work in [29, 30].
- Algebraic factorization or constrained algebraic resubstitution with complement preserves hazard-free robust path delay fault testability [20, 21].
- Algebraic factorization with [14] or without [15, 13] complement is hazard-non-increasing. In other words, if the original two-level logic is hazard-free then the multi-level logic obtained after algebraic factorization will remain hazard-free.

The first two results above are applicable when the original circuit to which we apply algebraic factorization is itself fully testable. The problem we face with asynchronous circuits is that these conditions are, in general, not met. However, as mentioned earlier, our aim would be to convert a two-level circuit, which may not be fully testable, to another intermediate circuit which is fully testable, and then apply algebraic factorization.

### 3.1 Starting from Non-Prime, but Irredundant Two-Level Logic

To motivate this case, let us consider an example first.

**Example 3.1.** Let us examine the sum of products expression:  $f = a\bar{c} + \bar{a}c + b\bar{c} + bcd + \bar{c}\bar{d}$  (which for simplicity of exposition is assumed to be hazard-free for some given set of input transitions). In this expression, all product terms are irredundant, i.e., testable for all stuck-at 0 faults. However, the product term  $bcd$  is non-prime because literal  $c$  in it is untestable for the stuck-at 1 fault. All other literals are testable for stuck-at 1 faults. From here on, we will denote a literal which is untestable for a stuck-at 1 fault by superscript “+”. If we are concerned about robust path delay fault testability, then we note that paths starting from literal  $\bar{c}$  in product  $b\bar{c}$ , from literal  $\bar{c}$  in product  $\bar{c}\bar{d}$ , and from literal  $c$  in product  $bcd$ , are not testable for delay faults in the corresponding two-level circuit. For the first case, the reason is that we need to make  $b = 1$  and make other product terms 0 without using  $c$  (since  $c$  either makes a rising or falling transition during the test, and hence does not have a fixed value). This condition can be derived from the necessary and sufficient conditions in [18]. The

above condition is not possible to satisfy because both  $a\bar{c}$  and  $\bar{a}c$  cannot be simultaneously made 0 without using  $c$ . The same reason is applicable to the other two cases. Paths starting from all the other literals can be shown to be robustly testable. From here on we will denote literals which are not robustly testable with the superscript “\*”. A literal which is not stuck-at 1 testable is obviously not robustly testable either. Thus, a literal with a “+” superscript is implicitly assumed to have a “\*” superscript too; however, the reverse is not true. With the above arguments, we can rewrite the sum of products as:  $f = a\bar{c} + \bar{a}c + b\bar{c}^* + bc^+d + \bar{c}^*\bar{d}$ . Assuming first that stuck-at testability is our only concern, we recognize that although a stuck-at 1 fault in literal  $c$  is untestable in product  $bcd$ , it is testable in product  $\bar{a}c$ . Factoring out  $c$  from these two product terms, we get the expression  $f = a\bar{c} + c(\bar{a} + bd) + b\bar{c}^* + \bar{c}^*\bar{d}$ , which is completely testable for all single stuck-at faults. If robust testability is the aim, then we recognize that although literal  $\bar{c}$  is robustly untestable in products  $b\bar{c}$  and  $\bar{c}\bar{d}$ , it is robustly testable in product  $a\bar{c}$ . Thus, by additional factoring we get  $\bar{c}(a + b + \bar{d}) + c(\bar{a} + bd)$ , which is completely robustly testable.  $\square$

Since we are using targeted algebraic factorization in the above example to achieve testability, the hazard-freedom of the original circuit is maintained. After the two-level expression is modified in the above fashion, further algebraic factorization based on the results mentioned earlier can be used to reduce area further while maintaining testability and its hazard-freedom. This example also shows how stuck-at and path delay faults can be treated in a similar way. The following theorem from [23] gives the condition under which merging of untestable and testable literals is possible for robust testability.

**Theorem 3.1.** Consider the switching expression  $f = \sum_{j=1}^{n_1} x_1x_2 \cdots x_m P_j + \sum_{j=1}^{n_2} R_j$ , where  $P_j$  and  $R_j$  are products of literals. Suppose that (a) in each product term in  $\sum_{j=1}^{n_1} x_1x_2 \cdots x_m P_j$ , all literals in  $P_j$  are robustly testable, (b) a product term in  $\sum_{j=1}^{n_1} x_1x_2 \cdots x_m P_j$  may not be robustly testable in a subset of literals of the set  $\{x_1, x_2, \dots, x_m\}$ , and (c) each literal in  $\{x_1, x_2, \dots, x_m\}$  is robustly testable in at least one product term in  $\sum_{j=1}^{n_1} x_1x_2 \cdots x_m P_j$ . Under these conditions, literals  $x_1, x_2, \dots, x_m$  are robustly testable when factored out from the first set of product terms obtaining the following modified expression:  $f = x_1x_2 \cdots x_m(\sum_{j=1}^{n_1} P_j) + \sum_{j=1}^{n_2} R_j$ . At the same time, all literals in  $\sum_{j=1}^{n_1} P_j$  remain robustly testable, and literals in  $\sum_{j=1}^{n_2} R_j$  retain their robust testability.

It turns out that one can obtain a very similar theorem for stuck-at faults too, as follows.

**Theorem 3.2.** Consider the switching expression  $f = \sum_{j=1}^{n_1} x_1x_2 \cdots x_m P_j + \sum_{j=1}^{n_2} R_j$ , where  $P_j$  and  $R_j$  are products of literals. Suppose that (a) in each product term in  $\sum_{j=1}^{n_1} x_1x_2 \cdots x_m P_j$ , all literals in  $P_j$  are testable for stuck-at faults, (b) a product term in  $\sum_{j=1}^{n_1} x_1x_2 \cdots x_m P_j$  may not be stuck-at testable in a subset of literals of the set  $\{x_1, x_2, \dots, x_m\}$ , and (c) each literal in  $\{x_1, x_2, \dots, x_m\}$  is

stuck-at testable in at least one product term in  $\sum_{j=1}^{n_1} x_1 x_2 \cdots x_m P_j$ . Under these conditions, literals  $x_1, x_2, \dots, x_m$  are stuck-at testable when factored out from the first set of product terms obtaining the following modified expression:  $f = x_1 x_2 \cdots x_m (\sum_{j=1}^{n_1} P_j) + \sum_{j=1}^{n_2} R_j$ . At the same time, all literals in  $\sum_{j=1}^{n_1} P_j$  remain stuck-at testable, and literals in  $\sum_{j=1}^{n_2} R_j$  retain their stuck-at testability.

**Proof:** The proof is very similar to the proof of Theorem 3.1 given in [23], and is hence omitted.  $\square$

A special case of the method based on Theorem 3.1 has been used in [26, 4] for obtaining robustly testable circuits where one path which is not robustly testable can be merged with another which is. However, such a method has not been applied to obtain single stuck-at testability before, starting from an initial two-level circuit which is not completely testable. For all the asynchronous circuit benchmarks we considered, when the two-level circuit was non-prime, but irredundant, the synthesis rules based on the above theorems were successful in obtaining completely testable solutions. When a sum of products has many literals which are either not stuck-at 1 testable or not robustly testable, then we need heuristics to apply these synthesis rules efficiently to arrive at a solution. These heuristics are the same as those given in [23], and hence are not repeated here. When the two-level circuit is redundant, then the above theorems will, in general, only be partially successful. Thus, in order to achieve complete testability in the multi-level circuit, we may need to add extra controllable inputs, as discussed in the next section. In some rare cases, even when the original two-level circuit is non-prime, but irredundant, the above synthesis rules may still not be applicable. When this happens, there are various options:

- If the synthesis rules were not successful with expression  $f$ , one can try to see their applicability to the sum of products of  $\bar{f}$  [4, 23]. The circuit realization of  $\bar{f}$  can be followed by an inverter to realize the original function. This does not affect the hazard-freedom of the realization [13].
- We can obtain another hazard-free sum of products expression if one exists, and try to apply the synthesis rules to it or its complement.

In the extremely rare cases where even the above options do not work, we would need to add extra controllable inputs to solve the problem, as illustrated by the following example.

**Example 3.2.** Consider the irredundant, but non-prime, sum of products  $f = \bar{c}^* \bar{d} + \bar{a}^* b + a \bar{c} + \bar{a} c + a^+ b c^+ d$ . Assume that this is a hazard-free expression for some specified set of input transitions. There are two literals in it which are not stuck-at 1 testable and four literals (including these two) which are not robustly path delay fault testable. Let us label the product terms consecutively as  $p_1, p_2, \dots, p_5$ . The testability problems in this expression cannot be solved with Theorems 3.1 or 3.2. Let us first

concentrate on stuck-at testability alone. The reason the two literals in  $p_5$  are stuck-at 1 untestable is due to the presence of  $p_2, p_3$  and  $p_4$ . For example, when we want to test literal  $a$  in  $p_5$ , we need to feed vector 0111. However, for this vector,  $p_2$  and  $p_4$  assume the value 1. Similarly, to test literal  $c$  in  $p_5$  we need to feed vector 1101, which makes  $p_3$  equal to 1. We denote this fact by a function  $P = P_5 = p_2p_3p_4$ . In order to make  $p_5$  testable, we need to make each of these product terms 0 with an extra controllable input  $t_1$ , which gives us the solution  $f = \bar{c}^* \bar{d} + t_1 \bar{a}^* b + t_1 a \bar{c} + t_1 \bar{a} c + abcd$ , which is completely testable for all single stuck-at faults (under normal operation,  $t_1 = 1$ ). In order to make it fully robustly testable for path delay faults, we can now use Theorem 3.1 to obtain  $f = \bar{c}(\bar{d} + t_1 a) + t_1 \bar{a}(b + c) + abcd$ . Consider now the complement of the above function, and assume that it has been made hazard-free for the same set of multiple input changes, as follows:  $\bar{f} = \bar{a}\bar{b}\bar{c}d + ab^+c\bar{d} + \bar{a}\bar{b}cd^+ + \bar{a}\bar{b}^+c\bar{d}^+$ . As before, let us label the product terms consecutively as  $p_1, \dots, p_4$ .  $p_2$  is not testable because of the presence of  $p_4$  (we denote this by a corresponding function  $P_2 = p_4$ ),  $p_3$  because of the presence of  $p_4$  (denoted by  $P_3 = p_4$ ) and  $p_4$  because of the presence of  $p_2$  and  $p_3$  (denoted by  $P_4 = p_2p_3$ ). To solve all the untestability problems, we need to remove all these causes for untestability. Thus, we derive a composite function  $P$  as  $P_2P_3P_4 = (p_4)(p_4)(p_2p_3)$  which reduces to  $p_2p_3p_4$ . Therefore,  $p_2, p_3$  and  $p_4$  would need extra controllable inputs. Denote the controllable inputs for product term  $p_i$  as  $t_{p_i}$ . In order to completely test  $p_2$  we have the constraint that  $(t_{p_2} \neq t_{p_4})$  because  $p_4$  needs to be made 0 with the help of the controllable input when we are testing  $p_2$ . Similarly, the constraint for completely testing  $p_3$  is  $(t_{p_3} \neq t_{p_4})$ , and for completely testing  $p_4$  it is  $(t_{p_4} \neq t_{p_2}$  and  $t_{p_4} \neq t_{p_3})$ . All the above constraints can be satisfied by  $t_{p_2} = t_{p_3} = t_1$  and  $t_{p_4} = t_2$ . Thus, the completely testable solution (for both stuck-at and delay faults) becomes  $\bar{f} = \bar{a}\bar{b}\bar{c}d + t_1 abc\bar{d} + t_1 \bar{a}\bar{b}cd + t_2 \bar{a}\bar{b}c\bar{d}$ . However, since this requires two extra inputs, the first solution for  $f$ , which required only one extra input, may be chosen.  $\square$

The above example was deliberately made pathological in order to show how the approach works. In general, requirement of an extra input for irredundant, but non-prime, expressions is very rare. We next formalize the approach outlined in the above example into a procedure, which we would apply only if none of the other options were successful in making the expression testable. This procedure is valid for both the stuck-at and robust path delay fault models. A method for determining what other product terms make a product term not robustly testable has been given in [23] and is illustrated later.

**Procedure 3.1.**

1. For the given hazard-free sum of products expression  $f$ , first identify and mark the literals and

product terms which are not testable under the specified fault model (stuck-at or robust path delay), and label the product terms as  $p_1, p_2, \dots, p_n$ .

2. For each product term  $p_i$  which has an untestable literal, identify the other product terms whose presence causes the untestability. Derive the corresponding function  $P_i$ .
3. Derive a composite function  $P$  by taking the logical AND of all such  $P_i$  functions, and simplify  $P$  into a minimal sum of products using laws from switching algebra.
4. Select the product term in  $P$  with the minimum number of literals which has not yet been processed (break any ties arbitrarily). These literals correspond to the product terms in  $f$  which will have an extra controllable input ANDed with them. Derive a set of constraints on these controllable inputs, and derive a minimal set of extra inputs which satisfy all these constraints.
5. If the number of extra inputs required in the previous step is more than 1, then consider other products in  $P$  and repeat the previous step until either a solution is found with just one extra input or all the products in  $P$  are exhausted. For the latter case, choose the solution which required the minimum number of extra inputs.
6. Modify  $f$  by inserting the extra controllable inputs as derived above. This expression is now completely testable under the given fault model.  $\square$

Procedure 3.1 can be applied to the hazard-free  $f$  first and then to the hazard-free  $\bar{f}$ . The one requiring fewer extra inputs can be selected. Once the two-level expression has been made testable, further algebraic factorization, as outlined at the beginning of this section, can be used to derive a testable and hazard-free multi-level circuit.

### 3.2 Starting from Redundant, but Prime Two-Level Logic

If the two-level expression is redundant, but prime, it means that some stuck-at 0 faults are untestable, but all stuck-at 1 faults are testable. We again motivate this case through an example first.

**Example 3.3.** Consider the expression  $f = \bar{y}^* \bar{z} + \bar{x}^\# \bar{y}^\# + \bar{x}^* z + xy$ , which is assumed to be hazard-free with respect to some specified set of input transitions. Here the superscript “#” denotes stuck-at 0 untestability of the literal (note that stuck-at untestability automatically implies that the literal is not robustly testable either). If we label the product terms as  $p_1, \dots, p_4$  as before, we find that  $p_2$  is redundant. All the stuck-at 1 faults are testable (hence the circuit is prime), however, some literals are not robustly testable, as also indicated with the “\*” superscript. Let us first concentrate on making the

circuit completely stuck-at testable. The reason  $p_2$  is redundant is because we cannot simultaneously make both  $p_1$  and  $p_3$  0 when we try to test  $p_2$ . Thus, to make  $p_2$  testable, making either  $p_1$  or  $p_3$  0 using a controllable input would be necessary. Using previously introduced notation, we denote this fact by  $P_2 = (p_1 + p_3)$ . Since there are no other stuck-at testability problems,  $P = P_2 = (p_1 + p_3)$ . If we decide to logically AND the extra input  $t_1$  with  $p_1$ , then the completely stuck-at testable solution becomes  $f = t_1 \bar{y} \bar{z} + \bar{x} \bar{y} + \bar{x} z + xy$ . ANDing  $t_1$  with  $p_3$  would be an equally valid solution. Suppose now that we would like to make  $f$  completely robustly testable. In [23], a method is given to identify the other product terms whose presence makes a given literal not robustly testable. For example, to test literal  $\bar{y}$  in  $p_1$  robustly, we need to make  $\bar{z} = 1$  and zero out the other product terms without using literal  $y$ .  $\bar{z} = 1$  makes  $p_3 = 0$ . However,  $p_2$  and  $p_4$  cannot simultaneously be made 0 without using literal  $y$ . Therefore,  $P_1 = (p_2 + p_4)$ . In  $p_2$ , literal  $\bar{x}$  is not robustly testable because of the condition  $(p_1 + p_3)$ , and literal  $\bar{y}$  is also not robustly testable coincidentally because of the same condition  $(p_1 + p_3)$ . Thus,  $P_2 = (p_1 + p_3)(p_1 + p_3)$  which reduces to  $(p_1 + p_3)$ . In  $p_3$ ,  $\bar{x}$  is not robustly testable and yields the condition  $P_3 = (p_2 + p_4)$ . Therefore,  $P = P_1 P_2 P_3 = (p_2 + p_4)(p_1 + p_3)(p_2 + p_4) = (p_2 + p_4)(p_1 + p_3) = p_1 p_2 + p_1 p_4 + p_2 p_3 + p_3 p_4$ . Since all product terms in  $P$  have the same number of literals, we can start with any one of them. Suppose we chose  $p_1 p_2$ . This means that  $p_1$  and  $p_2$  would have extra controllable inputs ANDed with them. Condition  $P_1$  implies the constraint that  $t_{p_1} \neq t_{p_2}$ . Similarly, condition  $P_2$  implies the constraint that  $t_{p_2} \neq t_{p_1}$  (which happens to be the same constraint as the previous one).  $P_3$  does not imply any constraints since no controllable input is going to be used with  $p_3$ . One way to satisfy the above constraints is by making  $t_{p_1} = t_1$  and  $t_{p_2} = t_2$ . Thus,  $f = t_1 \bar{y} \bar{z} + t_2 \bar{x} \bar{y} + \bar{x} z + xy$  is completely robustly testable. Any other choice of product term from  $P$  would also have led to two extra inputs. Before implementing this solution, one can check if the hazard-free expression for  $\bar{f}$  can be made fully testable with just one or no extra inputs.  $\square$

The above example illustrates that Procedure 3.1 remains valid for tackling redundant, but prime, sum of products expressions too. It is just that the  $P_i$  functions have a somewhat different look in this case compared to the irredundant, but non-prime, case. Also, the same procedure is applicable for deriving both stuck-at and robustly testable circuits, the only difference again being that the  $P_i$  functions are different in the two cases.

### 3.3 Starting from Redundant and Non-Prime Two-Level Logic

Although Procedure 3.1 can simultaneously solve the untestability problems arising from both redundancy and non-primeness, the following approach would be better in reducing the number of extra

inputs.

- We first add extra inputs to solve the redundancy problem only, through Procedure 3.1, and try to solve the remaining problem of either non-primeness or robust untestability using Theorem 3.1 or 3.2, as the case may be.
- Only if Theorem 3.1 or 3.2 is not fully successful in solving the residual testability problems, do we attack the problems of redundancy and non-primeness (or robust untestability) simultaneously through Procedure 3.1.

The following example will illustrate the reasons for the above approach.

**Example 3.4.** Consider the expression  $f = xy + \bar{x}z + y^\#z^\# + wy\bar{z}^+ + v\bar{z}$ . Label the product terms consecutively as  $p_1, \dots, p_5$ , as before. Here  $p_3$  is redundant (meaning both its literals are untestable for a stuck-at 0 fault) and literal  $\bar{z}$  in  $p_4$  is non-prime. Also, these are the only three literals which are robustly untestable. Let us concentrate on the stuck-at testability problem first. To solve the redundancy problem only, we derive  $P = P_3 = (p_1 + p_2)$ . Since these do not imply any constraints, we can solve this problem by ANDing either  $p_1$  or  $p_2$  with an extra controllable input  $t_1$ . At the same time, we realize that Theorem 3.2 can make literal  $\bar{z}$  in  $p_4$  testable by factoring this literal out from  $p_4$  and  $p_5$ . Therefore, a completely stuck-at testable solution is  $f = t_1xy + \bar{x}z + yz + \bar{z}(wy + v)$ . If we had tried to solve both the redundancy and non-primeness problems simultaneously, we would have obtained  $P_4 = (p_1 + p_2)p_3$  and  $P = P_3P_4 = p_1p_3 + p_2p_3$ . Suppose the first product term  $p_1p_3$  in  $P$  is used to find a solution, then the only constraint from  $P_3$  would imply that  $t_{p_3} \neq t_{p_1}$ . This means that two extra inputs would be required. Similarly, using product term  $p_2p_3$  also would lead to two extra inputs. This is clearly an overkill for solving the stuck-at untestability problems in  $f$ . Now let us switch our attention to robust testability. For this case,  $P_3 = (p_1 + p_2)(p_1 + p_2) = (p_1 + p_2)$  and  $P_4 = (p_1 + p_2)p_3$ . If we solve the testability problem in just the redundant term  $p_3$  first, and then apply Theorem 3.1 to the robustly untestable literal in  $p_4$ , we would end up with the same solution as above with just one extra input. However, if we try to solve all the robust untestability problems in  $f$  simultaneously, then  $P = P_3P_4 = p_1p_3 + p_2p_3$ , which is the same  $P$  function as obtained for the stuck-at case. Thus, in this case, again two extra inputs would be required.  $\square$

### 3.4 Starting from Irredundant and Prime Two-Level Logic

We finally come to the fourth and simplest case. For such a case, there can be no stuck-at testability problem. However, a prime and irredundant sum of products may still have robust untestabilities in

it. This can be solved as before, i.e., we first try to apply Theorem 3.1 to these untestabilities, failing which we switch to Procedure 3.1.

## 4 Experimental Results

We have applied our synthesis-for-testability algorithms to a number of examples. Table 1 gives the area and delay for a variety of benchmark circuits, using a standard cell implementation, including a second-level cache controller (*cache-ctrl*), controllers from Hewlett-Packard Laboratories for an experimental infrared communication chip (*it-control*, *two-ticks-if*, *rf-control*, *mod-sd-control*) and a routing chip (*sbuf-read-ctl*, *sbuf-send-ctl*, *pe-send-ifc*), an A-to-D converter (*chu-ad-opt-e*, *vanbek-ad-opt*), control for distributed mutual exclusion (*dme-e*, *dme-fast-e*), and other benchmarks (*bad-merge*). The “base” case refers to optimized multi-level hazard-free circuits, where testability is not a concern. Under “stuck-at testable” we give the area and delay of a hazard-free and completely stuck-at testable optimized multi-level circuit, and the number of extra inputs (*EI*) required. Under “robustly testable” we give the area and delay of a hazard-free and completely robustly path delay fault testable optimized multi-level circuit, and the number of extra inputs required. The results were obtained after technology mapping in the SIS [31] logic synthesis system with the *stdcell2\_2.genlib* CMOS cell library. The area is a relative figure obtained from the layout of the standard cells. The delay represents the critical path delay through the circuit, and is measured in *nanoseconds*. As indicated in the table, extra inputs were not required in any case. This is because we were able to obtain irredundant, but possibly non-prime, two-level logic as the starting point for each of these multi-level circuits. Also, the area overhead for testability is either zero or very small. Zero overhead is obtained when the general algebraic factorization used for the base case coincides with the targeted algebraic factorization used for the testable cases.

## 5 Conclusions

In this paper, we presented an efficient and complete method for synthesizing hazard-free implementations of asynchronous circuits which are also completely testable under the stuck-at and robust path delay fault models. Although, theoretically, to make the method complete, we may need to introduce extra control inputs in some rare cases, for none of the practical circuits that we encountered did we need an extra input. The area and delay overheads for obtaining the testable solutions were also shown to be minimal.

Table 1: Experimental results

circuit	base		stuck-at testable			robustly testable		
	area	delay	area	delay	$EI$	area	delay	$EI$
bad-merge	328	5.4	328	5.4	0	368	5.4	0
chu-ad-opt-e	160	3.0	160	3.0	0	160	3.0	0
dme-e	232	4.2	232	4.2	0	232	4.2	0
dme-fast-e	296	4.4	296	4.4	0	296	4.4	0
it-control	592	5.4	592	5.4	0	592	5.4	0
mod-sd-control	1352	8.0	1352	8.0	0	1352	8.0	0
pe-send-ifc	592	6.2	592	6.2	0	592	6.2	0
rf-control	408	4.4	408	4.4	0	408	4.4	0
sbuf-read-ctl	216	4.2	216	4.2	0	216	4.2	0
sbuf-send-ctl	352	4.4	352	4.4	0	352	4.4	0
two-ticks-if	104	2.8	104	2.8	0	104	2.8	0
vanbek-ad-opt	160	4.2	160	4.2	0	160	4.2	0
cache-ctrl	6192	12.0	6264	11.8	0	6808	11.4	0

## Acknowledgments

Thanks to Profs. Robert Brayton (UC Berkeley) and Luciano Lavagno (University of Torino) for first posing the problem of non-prime and redundancy minimization during hazard-free two-level logic optimization. Thanks to S. Bhatia and S. Srinivasan for help with generating the experimental results.

## References

- [1] P. A. Beerel and T. H.-Y. Meng, "Semi-modularity and self-diagnostic asynchronous control circuits," in *Proc. Conf. Adv. Res. VLSI*, Mar. 1991.
- [2] A. J. Martin and P. J. Hazewindus, "Testing delay-insensitive circuits," in *Proc. Conf. Adv. Res. VLSI*, Mar. 1991.
- [3] P. A. Beerel and T. H.-Y. Meng, "Testability of asynchronous timed control circuits," in *Proc. Design Automation Conf.*, pp. 446-451, June 1991.
- [4] K. Keutzer, L. Lavagno, and A. L. Sangiovanni-Vincentelli, "Synthesis for testability techniques for asynchronous circuits," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 1569-1577, Dec. 1995.
- [5] A. Khoche and E. Brunvand, "Testing micropipelines," in *Proc. Int. Symp. Adv. Res. Asynchronous Circuits & Systems (Async94)*, pp. 239-246, Nov. 1994.
- [6] M. Roncken, "Partial scan test for asynchronous circuits illustrated on a DCC error corrector," in *Proc. Int. Symp. Adv. Res. Asynchronous Circuits & Systems (Async94)*, pp. 247-256, Nov. 1994.
- [7] L. Lavagno, M. Kishinevsky, and A. Liroy, "Testing redundant asynchronous circuits by variable phase splitting," in *Proc. European Design Automation Conf.*, Sept. 1994.

- [8] C.-L. Wey, M.-D. Shieh, and P. D. Fisher, "ASLCScan: A scan design technique for asynchronous sequential logic circuits," in *Proc. Int. Conf. Computer Design*, pp. 159-162, Oct. 1993.
- [9] E. Eichelberger and T. W. Williams, "A logical design structure for LSI testing," in *Proc. Design Automation Conf.*, pp. 462-468, June 1977.
- [10] Y. K. Malaiya and R. Narayanaswamy, "Testing for timing faults in synchronous sequential integrated circuits," in *Proc. Int. Test Conf.*, pp. 560-571, Oct. 1983.
- [11] S. M. Nowick and D. L. Dill, "Exact two-level minimization of hazard-free logic with multiple-input changes," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 986-997, Aug. 1995.
- [12] J. G. Bredeson and P. T. Hulina, "Elimination of static and dynamic hazards for multiple input changes in combinational switching circuits," *Information & Control*, vol. 20, pp. 114-224, 1972.
- [13] D. Kung, "Hazard-non-increasing gate-level optimization algorithms," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1992.
- [14] D. Kung, Personal communication, Oct. 1993.
- [15] S. H. Unger, *Asynchronous Sequential Switching Circuits*, Wiley-Interscience, New York, NY, 1969.
- [16] E. J. McCluskey, *Logic Design Principles*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
- [17] S. Kundu, S. M. Reddy, and N. K. Jha, "Design of robustly testable combinational logic circuits," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 1036-1048, Aug. 1991.
- [18] C. J. Lin and S. M. Reddy, "On delay fault testing in logic circuits," *IEEE Trans. Computer-Aided Design*, pp. 694-703, Sept. 1987.
- [19] S. M. Reddy, C. J. Lin, and S. Patil, "An automatic test pattern generator for the detection of path delay faults," in *Proc. Int. Conf. Computer-Aided Design*, pp. 284-287, Nov. 1987.
- [20] S. Devadas and K. Keutzer, "Synthesis and optimization procedures for robustly delay-fault testable combinational logic circuits," in *Proc. Design Automation Conf.*, pp. 221-227, June 1990.
- [21] M. J. Bryan, S. Devadas, and K. Keutzer, "Testability preserving circuit transformations," in *Proc. Int. Conf. Computer-Aided Design*, pp. 456-459, Nov. 1990.
- [22] A. Pramanick, S. M. Reddy and S. Sengupta, "Synthesis of combinational logic circuits for path delay fault testability," in *Proc. Int. Symp. Circuits & Systems*, pp. 3105-3108, May 1990.
- [23] N. K. Jha, I. Pomeranz, S. M. Reddy, and R. J. Miller, "Synthesis of multi-level combinational circuits for complete robust path delay fault testability," in *Proc. Int. Symp. Fault-Tolerant Computing*, pp. 280-287, July 1992.
- [24] I. Pomeranz and S. M. Reddy, "Achieving complete delay fault testability by extra inputs," in *Proc. Int. Test Conf.*, pp. 273-281, Oct. 1991.
- [25] W.K. Lam, A. Saldanha, R.K. Brayton and A.L. Sangiovanni-Vincentelli, "Delay fault coverage, test set size, and performance trade-offs," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 32-44, Jan. 1995.
- [26] A. K. Pramanick, "Delay testing and design for testability for delay faults in combinational logic circuits," Ph.D. thesis, Dept. of Electrical & Computer Engg., The Univ. of Iowa, May 1991.

- [27] R. L. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Trans. Computer-Aided Design*, pp. 727-750, Sept. 1987.
- [28] J. Rajski and J. Vasudevamurthy, "The testability preserving concurrent decomposition and factorization of Boolean expressions," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 778-793, June 1992.
- [29] G. Hachtel *et al.*, "On properties of algebraic transformations and the multifault testability of multilevel logic," in *Proc. Int. Conf. Computer-Aided Design*, pp. 422-425, Nov. 1989.
- [30] R. Dandapani and S. M. Reddy, "On the design of logic networks with redundancy and testability considerations," *IEEE Trans. Comput.*, vol. C-23, pp. 1139-1149, Nov. 1974.
- [31] E. M. Sentovich *et al.*, "Sequential circuit design using synthesis and optimization," in *Proc. Int. Conf. Computer Design*, pp. 328-333, Oct. 1992.