Exact Two-Level Minimization of Hazard-Free Logic with Multiple-Input Changes*

Steven M. Nowick
Department of Computer Science
Columbia University
New York, NY 10027
nowick@cs.columbia.edu

David L. Dill
Department of Computer Science
Stanford University
Stanford, CA 94305
dill@hohum.stanford.edu

Abstract

This paper describes a new method for exact hazard-free logic minimization of Boolean functions. Given an incompletely-specified Boolean function, the method produces a minimum-cost sum-of-products implementation which is hazard-free for a given set of multiple-input changes, if such a solution exists. The method is a constrained version of the Quine-McCluskey algorithm. It has been automated and applied to a number of examples. Results are compared with results of a comparable non-hazard-free method (espresso-exact). Overhead due to hazard-elimination is shown to be negligible.

1 Introduction

There has been renewed interest in asynchronous design because of the potential benefits of improved system performance, modular design, and avoidance of clock skew [28, 15, 22, 38, 16, 23, 10, 8, 3, 24, 37, 2]. However, a major obstacle to correct asynchronous design is the problem of *hazards*, or undesired glitches in a circuit. The elimination of all hazards from asynchronous designs is an important and difficult problem. Many existing design methods do not guarantee freedom from all hazards; other methods are limited by harsh restrictions on input behavior (single-input changes only) or implementation style (the use of large, slow inertial delays) to insure correct operation.

The focus in this paper is on a particular class of hazards: hazards in combinational logic. The design of hazard-free combinational logic is critical to the correctness of most asynchronous designs. Our goal is the synthesis of combinational logic which *avoids all combinational hazards for a given set of multiple-input changes.*

In the following presentation, we are interested in circuits which function correctly assuming arbitrary gate and wire delays. We do not consider circuits which depend on bounded delay assumptions for correct operation or which make use of added delay elements to fix or filter out glitches.

The contribution of this paper is a solution to an open problem in logic synthesis: Given an incompletely-specified Boolean function and a set of multiple-input changes, produce an *exact minimized two-level implementation* which is hazard-free for every specified multiple-input change, *if*

*This work was supported by the Semiconductor Research Corporation, Contract no. 92-DJ-205, and by the Stanford Center for Integrated Systems, Research Thrust in Synthesis and Verification of Multi-Module Systems.

such a solution exists. Our method is a constrained version of the Quine-McCluskey algorithm [19]. The method has been automated and applied to a number of examples. Results are compared with results of a comparable non-hazard-free method (*espresso-exact* [33]). Overhead due to hazard-elimination is shown to be negligible.

Our method solves a general combinational synthesis problem which arises in many asynchronous sequential applications. Indeed, the method has already been incorporated into synthesis programs for three distinct asynchronous design styles: the *locally-clocked* [25, 29, 28, 31], *3D* [38] and *UCLOCK* [26] methods.

1.1 Previous Work.

Much of the original work on combinational hazards was limited to the case of *single-input changes*. Methods for detecting and eliminating combinational hazards for single-input changes were developed by Huffman, McCluskey and Unger and are described in [35].

Eichelberger [11] considered a particular class of combinational hazards for *multiple-input changes*: *static* hazards. There are two types of static hazards: *function* and *logic* hazards (see Section 3 below for definitions). Function hazards cannot be removed; logic hazards can be eliminated by using a sum-of-products implementation containing every prime implicant. Others have developed improved algorithms for selective static hazard elimination.

Dynamic combinational function and logic hazards for multiple-input changes were identified in [35, 7, 4]. Unger [35], Bredeson and Hulina [7], Bredeson [6], Beister [4] and Frackowiak [12] presented conditions to avoid dynamic logic hazards in two-level and multi-level circuits during multiple-input changes. They also indicate that these conditions cannot always be satisfied.

No general two-level hazard-free logic minimization method has been proposed for incompletely-specified functions allowing multiple-input changes. McCluskey [18] presented an exact hazard-free two-level minimization algorithm for single-input changes. Several methods have been proposed for the multiple-input change case, but each has limitations. Bredeson and Hulina [7] described an algorithm which produces hazard-free *sum-of-products* implementations for multiple-input changes. However, their algorithm uses sequential storage elements to implement combinational functions, where storage elements must satisfy special timing constraints.

Bredeson [6] later presented an algorithm for hazard-free *multi-level* implementations of combinational functions with multiple-input changes without storage elements. However, the algorithm does not demonstrate optimality, assumes a fully-specified function, and attempts to eliminate hazards even for unspecified transitions; in practice, results may be far from optimal. The algorithm also cannot generate certain minimal two-level implementations (if they include non-prime implicants; to be discussed later).

Closer to our work, Frackowiak [12] presented two exact hazard-free minimization algorithms for two-level implementations allowing multiple-input changes in a fully-specified function. Both algorithms eliminate dynamic hazards in specified transitions. However, the first ignores static hazards while the second attempts to eliminate static hazards even for unspecified transitions. Therefore, results may be either hazardous or suboptimal.

2 Definitions.

The following definitions are taken from [32, 33] with minor modifications (see also [5, 19]). We consider only single-output functions having binary input and output variables.

Define sets $P = \{0,1\}$ and $B = \{0,1,*\}$. A *Boolean function*, f, of n variables, $x_1, x_2, \ldots, x_n$, is defined as a mapping: f: $P^n \rightarrow B$. The value "*" in B represents a *don't-care* value of the function.

Each element in the domain P^n of function f is called a *minterm* of the function. A minterm is also called an *input state* of the function.

The *ON-set* of a function is the set of minterms for which the function has value 1. The *OFF-set* is the set of minterms for which the function has value 0. The *DC-set* (don't-care set) is the set of minterms for which the function has value "*".

A *literal* is a Boolean function of n variables, $x_1, x_2, \ldots, x_n$, defined as follows. Each variable, x_i, has three corresponding *literals*: x_i, $\overline{x_i}$ and x_i^*. Literal $x_i = 1$ for a minterm if and only if variable x_i in the minterm has value 1; literal $\overline{x_i} = 1$ if and only if x_i has value 0; and $x_i^* = 1$ if x_i has value 0 or 1 (*don't-care literal*).

A *product* term is a Boolean product (AND) of literals. If a product term evaluates to 1 for a given minterm, the product term is said to *contain* the minterm.

A *cube* is a set of minterms which can be described by a product term.

A *sum-of-products* represents a set of products; it is denoted by Boolean sum of product terms. A sum-of-products is said to contain a minterm if some product in the set contains the minterm.

A product Y *contains* a product X ($X \subseteq Y$) if the cube for X is a subset of the cube for Y. The *intersection* of products X and Y is the set of minterms contained in the intersection of the corresponding cubes.

An *implicant* of a function is a product term which contains no minterm in the function's OFF-set. A *prime implicant* of a function is an implicant contained in no other implicant of the function. An *essential prime implicant* is a prime implicant containing an ON-set minterm contained in no other prime implicant.

A *cover* of a Boolean function is a sum-of-products which contains all of the minterms of the ON-set of the function and none of the minterms of the OFF-set. A cover may also include minterms from DC-set. A standard cost function for covers is assumed where each implicant has the same cost.[1]

The *two-level logic minimization problem* is to find a minimum-cost cover of a function.

3 Background and Problem Statement.

3.1 Circuit and Delay Model.

This paper considers combinational circuits having arbitrary finite gate and wire delays [21, 18]. Each wire is modelled as a connection with an attached delay element, describing the total wire delay. Each gate is modelled as an instantaneous Boolean operator with a delay element attached to its output wire, describing the total gate delay. The delays may have arbitrary but finite values. Since delay elements are attached only to wires, this model has been called the *unbounded wire delay model*.

A *pure delay* model is assumed as well (see [4]). A pure delay can delay the propagation of a waveform, but does not otherwise alter it. That is, unlike the *inertial delay* model, this model conservatively assumes that glitches are not filtered out by delays on gates and wires [4].

A *delay assignment* is an assignment of fixed, finite delay values to every gate and wire in a circuit.

[1] The cost function can be generalized for single-output functions to include literal-count as a secondary cost (see also discussion in [32], page 14).

3.2 Multiple-Input Changes.

A *transition cube* (*cf.* [4, 6]) is a cube with a *start point* and an *end point*. Given input states A and B, the transition cube [A,B] from A to B has start point A and end point B and contains all minterms that can be reached during a transition from A to B. More formally, if A and B are described by products, with i-th literals A_i and B_i, respectively, then the i-th literal for the product of [A,B] is the Boolean function $A_i + B_i$. Note that the sum of complementary literals x and $\overline{x}$ is the don't-care literal, x^*.

The *open transition cube* (A,B) from A to B is defined as: [A,B] - B.

A *multiple-input change* or *input transition* from input state A to B is described by transition cube [A,B]. There are three properties which characterize a multiple-input change. First, inputs change *concurrently*, in any order and at any time. Equivalently, a simultaneous input change can be assumed, since inputs may be skewed arbitrarily by wire delays (see above). Second, inputs change *monotonically*: each input changes value at most once. And, finally, the input change occurs in *fundamental mode*: once a multiple-input change occurs, no further input changes may occur until the circuit has stabilized.

An input transition occurs during a *transition interval*, $t_I \leq t \leq t_F$, where inputs change at time t_I and the circuit returns to a steady state at time t_F [4].

An input transition from input state A to B for a Boolean function f is a *static transition* if f(A)=f(B); it is a *dynamic transition* if f(A)$\neq f$(B). In this paper, we consider only static and dynamic transitions where f is fully defined; that is, for every $X \in [A,B]$, $f(X) \in \{0,1\}$.

3.3 Function Hazards

A function f which does not change monotonically during an input transition is said to have a *function hazard* in the transition. The following definitions are from Bredeson and Hulina [7] (see also [11, 6, 4, 20]).

Definition. A Boolean function f contains a *static function hazard* for the input transition from A to C if and only if:

1. $f(A) = f(C)$, and
2. there exists some input state $B \in [A,C]$ such that $f(A) \neq f(B)$.

Definition. A Boolean function f contains a *dynamic function hazard* for the input transition from A to D if and only if:

1. $f(A) \neq f(D)$.
2. There exist a pair of input states B and C ($A \neq B$, $C \neq D$) such that
   (a) $B \in [A,D]$ and $C \in [B,D]$ and
   (b) $f(B) = f(D)$ and $f(A) = f(C)$.

If a transition has a function hazard, *no* implementation of the function is guaranteed to avoid glitches during the transition, assuming arbitrary gate and wire delays [11, 7]. Therefore, we consider only transitions which are free of function hazards (see [11, 6, 4]).

Example. The function f of figure 1 has a static function hazard for the multiple-input change from i to k, since $f(i) = f(k) = 1$, $f(j) = 0$, and $j \in [i,k]$. The function has a dynamic function hazard for the transition from g to j, since $f(g) = 1$, $f(j) = 0$, $h \in [g,j]$, $i \in [h,j]$, $f(g) = f(i) = 1$ and $f(h) = f(j) = 0$. The input transition from k to m is free of static function hazards, and the input transition from n to p is free of dynamic function hazards. □

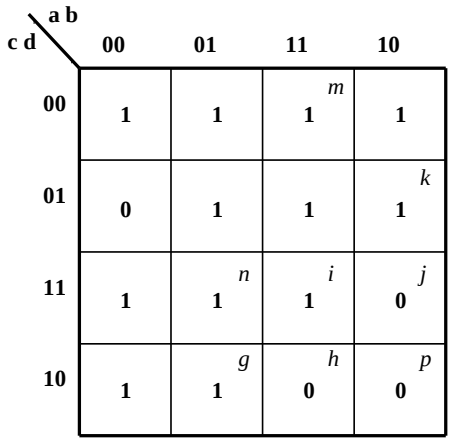


Figure 1: Boolean Function with Function Hazards

3.4 Logic Hazards.

If f is free of function hazards for a transition from input A to B, it may still have hazards due to possible delays in the actual logic realization [35, 7, 4]. In the following, a signal is called "monotonic" during a transition interval if it changes at most once (*i.e.,* weakly monotonic).

Definition. A combinational circuit for a function f contains a *static logic hazard* for the input transition from minterm A to minterm B if and only if:

1. f is function-hazard-free for the input transition.
2. $f(A) = f(B)$.
3. For some delay assignment, the circuit's output is not monotonic during the transition interval.

Definition. A combinational circuit for a function f contains a *dynamic logic hazard* for the input transition from minterm A to minterm B if and only if:

1. f is function-hazard-free for the input transition.
2. $f(A) \neq f(B)$.
3. For some delay assignment, the circuit's output is not monotonic during the transition interval.

3.5 Two-Level Hazard-Free Logic Minimization Problem.

The two-level hazard-free logic minimization problem can now be stated as follows:
Given:

A Boolean function f, and a set, T, of *specified* function-hazard-free (static and dynamic) input transitions of f.

Find:

A minimum-cost cover of f whose AND-OR implementation is free of logic hazards for every input transition $t \in T$.

4 Conditions for a Hazard-Free Transition.

We now describe conditions to insure that a sum-of-products implementation is hazard-free for a given input transition. Assume that [A,B] is the transition cube corresponding to a *function-hazard-free* transition from input state A to B for a combinational function f. In the following discussion, we assume that C is any cover of f implemented in AND-OR logic. (It is assumed that no product contains a pair of complementary literals, otherwise additional hazards are possible; see [35].)

The following lemmas present necessary and sufficient conditions to insure that the AND-OR implementation of f has *no logic hazards* for the given transition:

Lemma 1. If f has a $0 \rightarrow 0$ transition in cube [A,B], then the implementation is free of logic hazards for the input change from A to B.

Lemma 2. If f has a $1 \rightarrow 1$ transition in cube [A,B], then the implementation is free of logic hazards for the input change from A to B *if and only if* [A,B] is contained in some cube of cover C.

The conditions for the $0 \rightarrow 1$ and $1 \rightarrow 0$ cases are symmetric. Without loss of generality, we consider only a dynamic $1 \rightarrow 0$ transition, where f(A)=1 and f(B)=0. (A $0 \rightarrow 1$ transition from A to B has the same hazards as a $1 \rightarrow 0$ transition from B to A.)

Lemma 3. If f has a $1 \rightarrow 0$ transition in cube [A,B], then the implementation is free of logic hazards for the input change from A to B *if and only if* every cube $c \in C$ intersecting [A,B] also contains A.

Proof. These results follow immediately from pp. 128-9 in [35] and Theorem 3.4 in [12]. See also, Theorem 4 in [7], Lemmas 2 and 3 in [6], Theorem 4.5 in [35], and [4]. □

Lemma 2 requires that in a $1 \rightarrow 1$ transition, *some* product holds its value at 1 throughout the transition. Lemma 3 insures that no product will glitch *in the middle* of a $1 \rightarrow 0$ transition: all products change value monotonically during the transition. In each case, the implementation will be free of hazards for the given transition.

An immediate consequence of Lemma 3 is that, if a dynamic transition is free of logic hazards, then every static sub-transition will be free of logic hazards as well:

Corollary 1. If f has a $1 \rightarrow 0$ transition from input state A to B which is hazard-free in the implementation, then, for every input state X$\in$[A,B) where f(X)=1, the transition subcube [A,X] is contained in some cube of cover C.

Proof. Since C is a cover of function f, there exists some cube $c \in C$ which contains X. Since f is hazard-free in the transition from A to B, then, by Lemma 3, cube c contains A as well; therefore c contains [A,X]. □

Corollary 2. If f has a $1 \rightarrow 0$ transition from input state A to B which is hazard-free in the implementation, then for every input state X$\in$[A,B) where f(X)=1, the static $1 \rightarrow 1$ transition from input state A to X is free of logic hazards.

Proof. Immediate from Lemma 2 and Corollary 1. □

Lemma 2 and Corollary 1 are used to define the covering requirement for a hazard-free transition. The cube [A,B] in Lemma 2 and the *maximal* subcubes [A,X] in Corollary 1 are called *required cubes*. These cubes define the ON-set of the function in a transition. Each required cube *must* be contained in some cube of cover C to insure a hazard-free implementation.

Lemma 3 constrains the cubes which may be included in a cover C. Each $1 \rightarrow 0$ transition cube is called a *privileged cube*, since no cube c in the cover may intersect it unless c contains its

start point. If a cube intersects a privileged cube but does not contain its start point, it *illegally intersects* the privileged cube and may not be included in the cover.

4.1 Hazard Example.

Figures 2 and 3 illustrate the conditions of Lemmas 2 and 3 and the two Corollaries. Each figure shows a multiple-input change where inputs a and b both change from 0 to 1. The transition is described by a state graph, which represents a portion of a Karnaugh map for the given transition. A state graph is used to describe transitions within a Karnaugh map. For example, if the top vertex in the state graph of Figure 2(a) corresponds to $abcd = 0000$ in the Karnaugh map of Figure 1, then the left, right and bottom vertices of the state graph would correspond to $abcd = 1000, 0100$ and 1100, respectively, in the Karnaugh map. In this case, the state graph indicates an input transition from $abcd = 0000$ to 1100.

Figure 2 shows covers for a $1 \rightarrow 1$ transition. The cover in figure 2(a) is hazardous. The cubes in the cover, M and N, correspond to AND-gates in the final AND-OR implementation. Initially, the M AND-gate is high and the N AND-gate is low. During the transition, the M AND-gate goes low and the N AND-gate goes high. For certain delays, however, the M AND-gate goes low before the N AND-gate goes high, and the circuit output glitches (see timing diagram).

The cover in figure 2(b) is hazard-free. As required by Lemma 2, the cover contains a product, P, which *completely contains* the transition cube. This product corresponds to an AND-gate in the implementation which *holds its value at 1* throughout the transition. Therefore, the circuit output will not glitch (see timing diagram).

Figure 3 shows covers for a $1 \rightarrow 0$ transition. The cover in figure 3(a) is hazardous: cubes R and S both illegally intersect the transition.

First, consider the sub-transition where only input a changes; the output must remain at 1. Therefore, this sub-transition is a $1 \rightarrow 1$ transition. However, no single product in the cover contains this sub-transition cube, so the *sub-transition* has a static hazard.

Alternatively, consider the case where input b changes first. This sub-transition is free of static hazards, since product Q covers the sub-transition. However, a problem remains for the dynamic transition: product R intersects the transition cube in the middle. This stray product corresponds to an AND-gate in the implementation. Initially, this AND-gate is low; it may then go high and then eventually it will go low. During a $1 \rightarrow 0$ transition, such a glitch on an AND-gate can propagate as a glitch to the AND-OR circuit output (see timing diagram).

The cover in figure 3(b) is hazard-free. Each $1 \rightarrow 1$ sub-transition is completely contained in a product of the cover and there are no stray cubes which intersect the transition in the middle (see timing diagram).

5 Hazard-Free Covers.

A *hazard-free cover* of function f is a cover of f whose AND-OR implementation is hazard-free for a *given set* of specified input transitions. It is assumed below that the set of input transitions completely defines the function: the circuit must be hazard-free for each specified transition, and for all other input states the function is undefined (*i.e.*, don't-care value).

The following new theorem describes all hazard-free covers for function f for a set of multiple-input transitions.

Theorem 1: Hazard-Free Covering Theorem. A sum-of-products C is a hazard-free cover for function f for all specified input transitions if and only if:

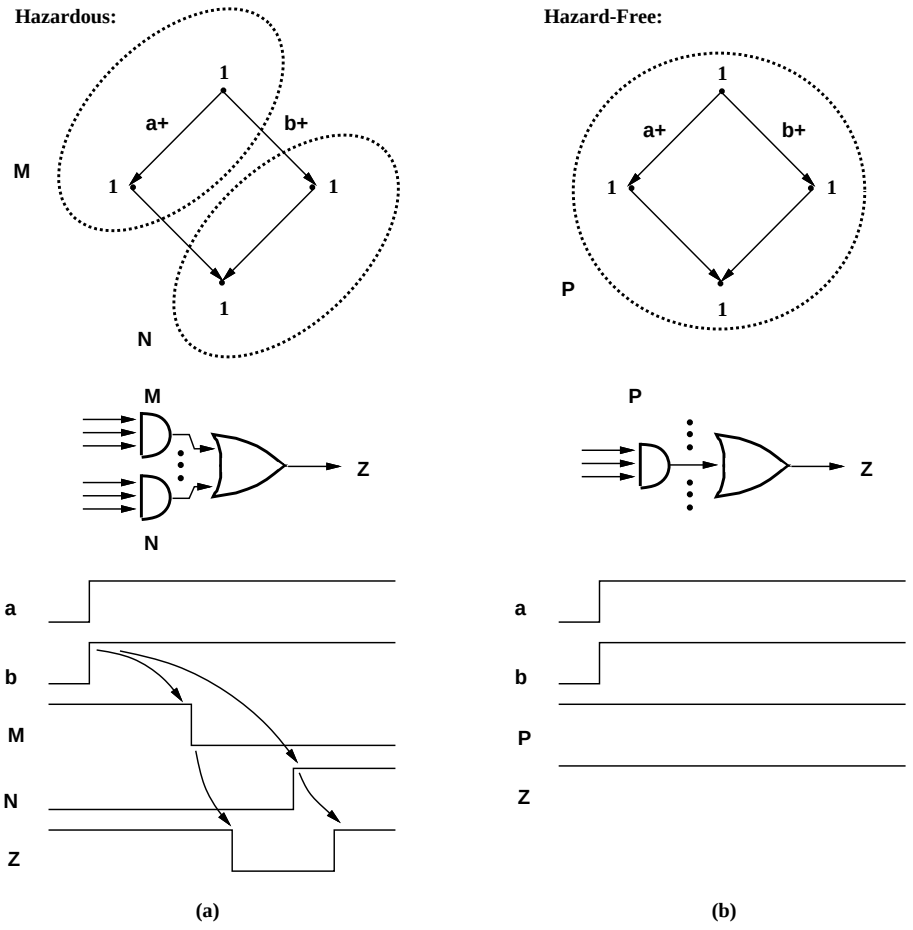


Figure 2: Hazardous and Hazard-Free Covers for a $1 \rightarrow 1$ Input Transition

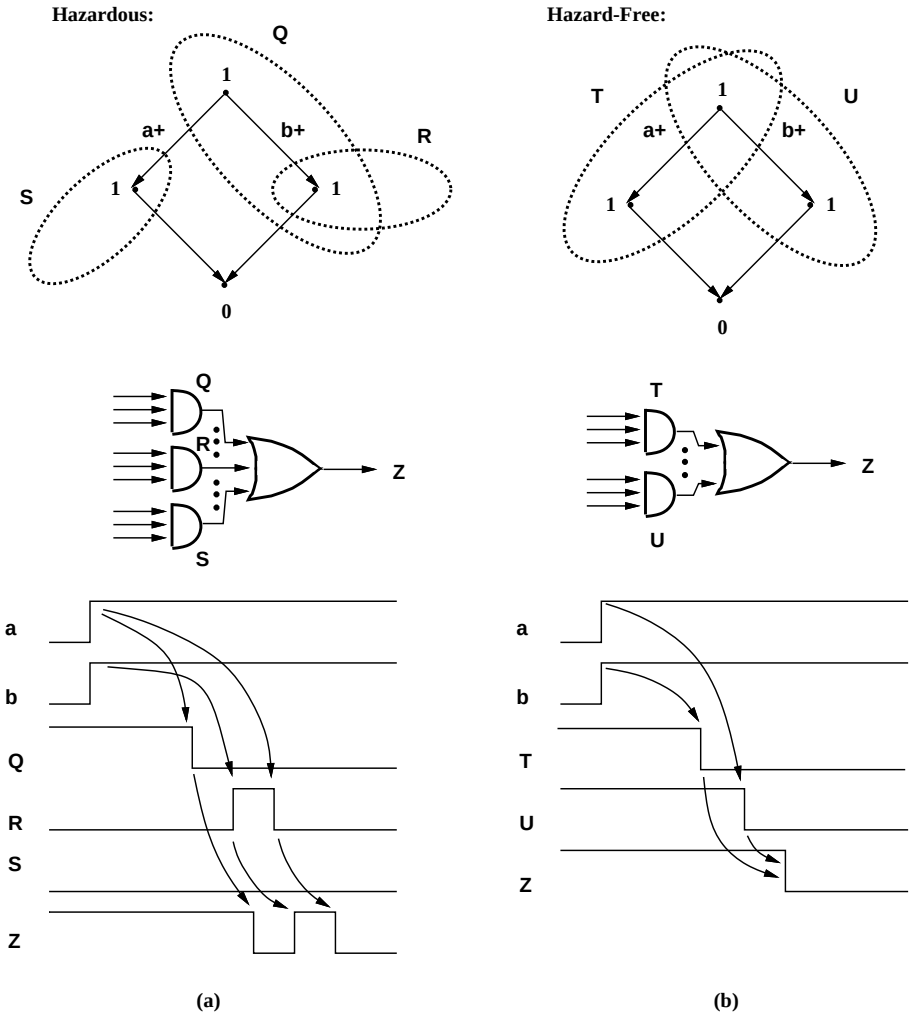


Figure 3: Hazardous and Hazard-Free Covers for a $1 \rightarrow 0$ Input Transition

(a) No cube of C intersects the OFF-set of f;

(b) Each *required cube* of f is contained in some cube of the cover, C; and

(c) No cube of C intersects any *privileged cube* illegally.

Proof. The result follows immediately from Lemmas 1–3, the Corollary, and the definitions of hazard-free cover, required cubes and privileged cubes. Conditions (a)-(c) insure that the function is covered correctly and hazard-free covering requirements are met for each specified input transition. □

Conditions (a) and (c) in Theorem 1 determine the implicants which may appear in a hazard-free cover of a Boolean function f. Condition (b) determines the covering requirement for these implicants in a hazard-free cover. Therefore, Theorem 1 precisely characterizes the covering problem for hazard-free two-level logic.

In general, the covering conditions of Theorem 1 may not be satisfiable for an arbitrary Boolean function and set of transitions (*cf.* [35, 4, 12]). This case occurs if conditions (b) and (c) cannot be satisfied simultaneously. It is discussed further in Section 8.

6 Exact Hazard-Free Logic Minimization.

Many exact logic minimization algorithms are based on the Quine-McCluskey algorithm [32, 33, 19]. The Quine-McCluskey algorithm solves the two-level logic minimization problem. It has three steps:

1. Generate the prime implicants of a function;
2. Construct a prime implicant table; and
3. Generate a minimal cover of this table.

Our two-level *hazard-free* logic minimization algorithm is based on a constrained version of the Quine-McCluskey algorithm. Only certain implicants may be included in a hazard-free cover, and covering requirements are more restrictive.

We base our approach on the Quine-McCluskey algorithm to demonstrate a simple solution to the hazard-free minimization problem. There now exist much more efficient algorithms than Quine-McCluskey [32, 33]; the hazard-elimination techniques described here can be applied to these methods as well.

Theorem 1(a) and (c) determine the implicants which may appear in a hazard-free cover of a Boolean function f. A *dynamic-hazard-free implicant* (or *dhf-implicant*) is an implicant which does not intersect any privileged cube of f illegally (cf. *DHA-Implikant* [12]). **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.

Interestingly, a prime implicant is not a dhf-prime implicant if it intersects a privileged cube illegally. A dhf-prime implicant may be a proper subcube of a prime implicant for the same reason.

Theorem 1(b) determines the covering requirement for a hazard-free cover of f: **every required cube of f must be covered**, that is, contained in some cube of the cover.

The *two-level hazard-free logic minimization problem* is therefore *to determine a minimum-cost cover of a function using only dhf-prime implicants where every required cube is covered.*

Our hazard-free Quine-McCluskey algorithm has the following steps:

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

```
Algorithm Make-Sets (set T of input transitions):
    req-set = {}; off-set = {}; priv-set = {};
    for each transition t of T
      A = start point of t; B = end point of t;
      t-cube = [A,B];

      case (t)
        0 → 0 transition:
          add t-cube to off-set;
        1 → 1 transition:
          add t-cube to req-set;
        1 → 0 (or 0 → 1) transition:
          add each maximal ON-set subcube to req-set;
          add each maximal OFF-set subcube to off-set;
          add t-cube and its start-point A to priv-set;
    return (req-set, off-set, priv-set).
```

Table 1: **Step 0:** Algorithm *Make-Sets*.

Step 0: Make Sets.

Before generating dhf-prime implicants, three sets must be constructed: the req-set, the off-set, and the priv-set. The req-set contains the required cubes for function *f*; it also defines the ON-set of the function. The off-set contains cubes precisely covering the OFF-set minterms. The priv-set is the set of privileged cubes along with their start points.

The sets are generated by a simple iteration through every specified transition of the given function, using Algorithm **Make-Sets** (see Table 1). If the function has a $0 \rightarrow 0$ change for a transition, the corresponding transition cube is added to the off-set. If the function has a $1 \rightarrow 1$ change, the transition cube is added to the req-set.

If the function has a $1 \rightarrow 0$ transition (or symmetrically, a $0 \rightarrow 1$ transition), then the maximal ON-set cubes are added to req-set and the maximal OFF-set cubes are added to off-set. In addition, the transition cube and its start point are also added to the priv-set, since this transition cube may not be intersected illegally. (A $0 \rightarrow 1$ transition from input state x to y is considered a $1 \rightarrow 0$ transition from input state y to x, so it has "start point" y.)

Step 1: Generate DHF-Prime Implicants.

The dhf-prime implicants for function *f* are generated in two steps. The first step generates the prime implicants of *f* from the req-set (which defines the on-set) and the off-set, using standard techniques [32, 33]. The second step transforms these prime implicants into dhf-prime implicants using algorithm **PI-to-DHF-PI**. This algorithm is a simpler version of Algorithm B in [12]. The algorithm iteratively refines the set of prime implicants until it generates the set of dhf-prime implicants. In practice, many prime implicants are also dhf-prime implicants (see Experimental Results). Also, there are fast existing algorithms to generate the the prime implicants of a function [33, 17].

Pseudo-code for the algorithm is given in Table 2. tmp-set is initialized to the set of prime implicants. The algorithm iteratively removes each implicant, p, from tmp-set. If p has no illegal

```
Algorithm PI-to-DHF-PI (pi-set, priv-set)
    tmp-set = pi-set; dhf-pi-set = {};

    while (not empty (tmp-set))
      remove a cube p from tmp-set;
      if (p has no illegal intersections with any cube of priv-set)
        add p to dhf-pi-set;
      else
              /* p illegally intersects a priv-set cube; */
              /* reduce p to avoid intersection */
        c = any cube of priv-set which p intersects illegally;
        for (each input variable v which appears as a don't-care
          literal in p and as literal v or v' in c)
          p-red = the maximal subcube of p where v is set
                to the complement of its value in c;
          add p-red to tmp-set;
    delete all cubes in dhf-pi-set contained in other cubes;
    return (dhf-pi-set).
```

Table 2: **Step 1:** Algorithm *PI-to-DHF-PI*.

intersections with any cube of priv-set, it is a dhf-implicant; it is placed in dhf-pi-set.

If p illegally intersects some privileged cube c in priv-set, then cube p is "split", or reduced, in every possible way by a single variable to avoid intersecting c. The reduced cubes are returned to tmp-set. In general, these reduced cubes may have *new* illegal intersections: a reduced cube, *p-red*, may illegally intersect a priv-set cube, c, even if p legally intersects c.

The algorithm terminates when tmp-set is empty. The resulting cubes in dhf-pi-set are all dhf-implicants. In addition, it is easily proved that the algorithm generates all dhf-prime implicants. Subcubes of other cubes in dhf-pi-set are removed; the result is the set of dhf-prime implicants.

As an optimization, we eliminate implicants that *contain no required cube*. If a dhf-implicant contains no required cubes, it can always be removed from a cover to yield a lower-cost solution. (Note that a dhf-prime implicant may intersect the ON-set and yet contain no required cube; see Experimental Results, Section 11.)

Step 2: Generate DHF-Prime Implicant Table.

A *dhf-prime implicant table* is constructed for the given function. The rows of the table are labelled with the *dhf-prime implicants* used to cover the columns. The columns are labelled with the *required cubes* which must be covered. The table sets up the two-level hazard-free logic minimization problem.

Step 3: Generate a Minimal Cover.

The dhf-prime implicant table is solved in three steps, using simple standard techniques. More sophisticated techniques can also be applied [32, 33, 5, 19].

First, *essential dhf-prime implicants* are extracted using standard techniques.

Second, the flow table is iteratively reduced. Rows and columns of the table may be removed using *row-dominance* and *column-dominance* operations, respectively. These operations may lead to further opportunities for *(secondary) essential dhf-prime implicant removal.* The operations are iterated until there is no further change.

Finally, if the table is still non-empty, a covering problem remains (*cyclic covering problem*). It is solved using an exhaustive algorithm called *Petrick's method.* Each column lists implicants which cover a required cube. The column is translated into a Boolean sum of rows; the covering problem for the table can be stated as a Boolean product of these sums. This product is multiplied out to generate all possible solutions. A minimal solution is then selected.

7 Hazard-Free Minimization Example.

The Karnaugh map from figure 1 is reproduced in figure 4 (the function is slightly modified from ex. 3.4 of [12]). Set $T = \{t_1, t_2, t_3, t_4\}$ contains four *specified* function-hazard-free input transitions. Each transition t_i is described by a transition cube C_i with start point m_i:

$$\begin{array}{lll} t_1\text{:} & m_1 = ab'c'd & C_1 = ac' \\ t_2\text{:} & m_2 = ab'cd' & C_2 = ab'c \\ t_3\text{:} & m_3 = a'bc'd' & C_3 = a'c' \\ t_4\text{:} & m_4 = a'bcd & C_4 = c \end{array}$$

The input transitions are indicated in figure 4(a). The start point of each transition is described by a dot, and its transition cube is described by a dotted circle.

Step 0: Make Sets.

The req-set, off-set and priv-set are generated using Algorithm *Make-Sets*, as illustrated in figure 4(b).

t_1:	*req-cube-1*	$= ac'$	t_4:	*req-cube-4*	$= a'c$
t_2:	*off-cube-1*	$= ab'c$		*req-cube-5*	$= bcd$
t_3:	*req-cube-2*	$= a'c'd'$		*off-cube-3*	$= acd'$
	req-cube-3	$= a'bc'$		*off-cube-4*	$= ab'c$
	off-cube-2	$= a'b'c'd$		*priv-cube-2*	$= c$
	priv-cube-1	$= a'c'$		*priv-start-2*	$= a'bcd$
	priv-start-1	$= a'bc'd'$			

The final sets produced by the algorithm are:

$$\begin{array}{ll} \text{req-set} = & \{ac', a'c'd', a'bc', a'c, bcd\}, \\ \text{off-set} = & \{ab'c, a'b'c'd, acd', ab'c\}, \\ \text{priv-set} = & \{\langle a'bc'd', a'c'\rangle, \langle a'bcd, c\rangle\}. \end{array}$$

Step 1: Generate DHF-Prime Implicants.

First, prime implicants are generated from the req-set and off-set:

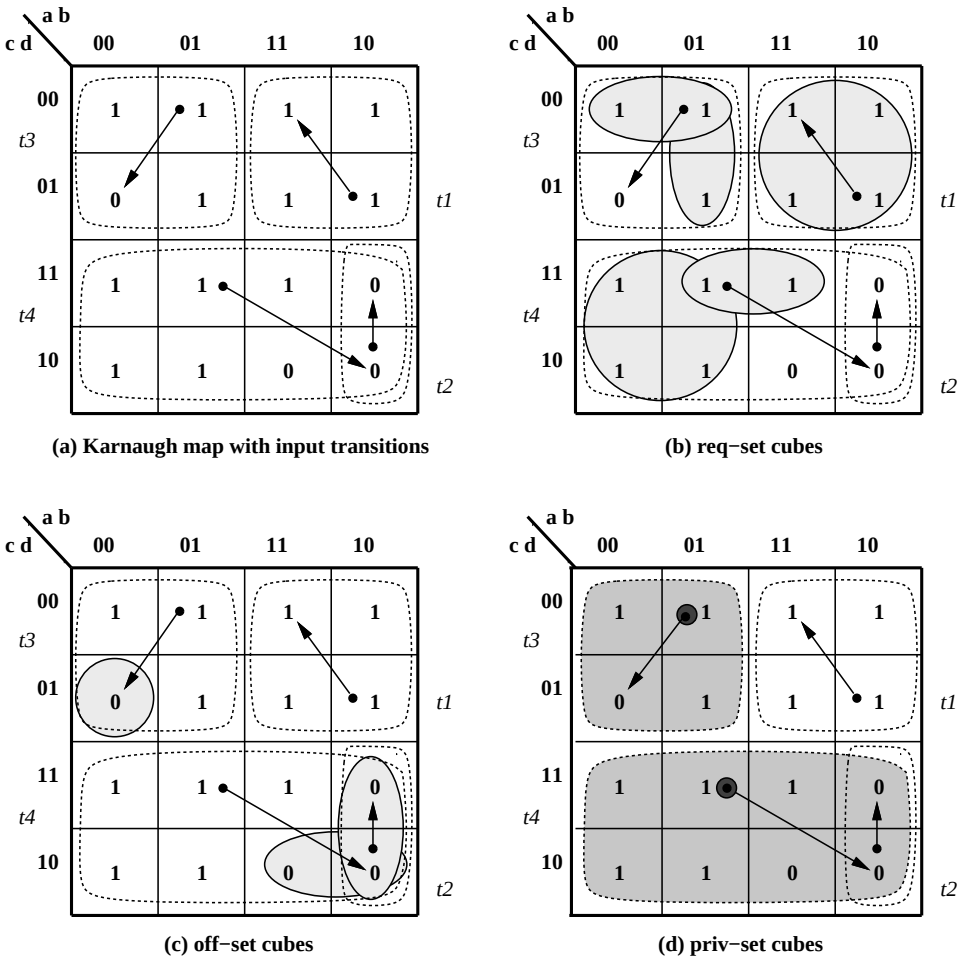


Figure 4: Hazard-Free Minimization Example: **Step 0**

$p_1 = c'd'$ $\quad$ $p_5 = a'c$
$p_2 = a'b$ $\quad$ $p_6 = bd$
$p_3 = bc'$ $\quad$ $p_7 = a'd'$
$p_4 = ac'$

The dhf-prime implicants are now produced using Algorithm *PI-to-DHF-PI*. The steps of the algorithm are illustrated in figure 5. Prime implicants p_1 through p_5 do not illegally intersect priv-set cubes *priv-cube-1* or *priv-cube-2*. As shown in figure 5(a), prime implicant p_1 intersects *priv-cube-1* and contains its start point. p_2 intersects both *priv-cube-1* and *priv-cube-2* and contains both start points. p_4 intersects neither priv-set cube. Similarly, p_3 and p_5 have no illegal intersections. These prime implicants are therefore dhf-prime implicants.

However, prime implicant p_6 illegally intersects *priv-cube-1*, since it intersects the cube ($bd \cap a'c' \neq \phi$) but does not contain its start point ($a'bc'd' \notin bd$; see Figure 5(b)). The algorithm splits p_6 into two subcubes: $p_{61} = bcd$ and $p_{62} = abd$ (see figure 5(c)). Cube p_{61} has no illegal intersections. However, p_{62} illegally intersects *priv-cube-2* (even though p_6 legally intersects *priv-cube-2*; see figure 5(b)). Cube p_{62} is again reduced to $p_{621} = abc'd$, which has no illegal intersections (see Figure 5(d)).

Similarly, prime implicant p_7 illegally intersects *priv-cube-2*, since $a'd' \cap c \neq \phi$ and $a'bcd \notin a'd'$ (see Figure 5(e)). Cube p_7 is reduced to $p_{71} = a'c'd'$, which has no illegal intersections (Figure 5(f)).

The resulting set of dhf-implicants is:

$$\{p_1, p_2, p_3, p_4, p_5, p_{61}, p_{621}, p_{71}\}.$$

After deleting cubes contained in other cubes, the final set of dhf-prime implicants is:

$$\{p_1, p_2, p_3, p_4, p_5, p_{61}\}.$$

Step 2: Generate DHF-Prime Implicant Table.

The dhf-prime implicant table for the example is shown in Table 3. The columns contain the required cubes generated in Step 0; the rows contain the dhf-prime implicants generated in Step 1.

Step 3: Generate a Minimal Cover.

A minimal cover is generated for the dhf-prime implicant table. The essential dhf-prime implicants are: p_1, p_4, p_5, and p_{61}. Either p_2 or p_3 can be selected to cover the remaining uncovered required cube, $a'bc'$. The function therefore has two minimal hazard-free covers, each containing 5 products: $\{p_1, p_4, p_5, p_{61}, p_2\}$ and $\{p_1, p_4, p_5, p_{61}, p_3\}$.

The latter cover is shown in figure 6(a). This cover is irredundant but *non-prime*, since it contains dhf-prime implicant p_{61} which is a proper subcube of prime implicant p_6.

A minimal *non-hazard-free* cover is shown in figure 6(b). The cover contains fewer products than the hazard-free cover, but has a logic hazard: prime implicant p_6 illegally intersects *priv-cube-1*. As a result, p_6 causes a dynamic hazard in the corresponding input transition, *t3*.

8 Existence of a Solution.

For certain Boolean functions and sets of transitions, the hazard-free covering problem has no solution [35, 4]. In this case, the dhf-prime implicant table will include at least one required cube which is not covered by any dhf-prime implicant.

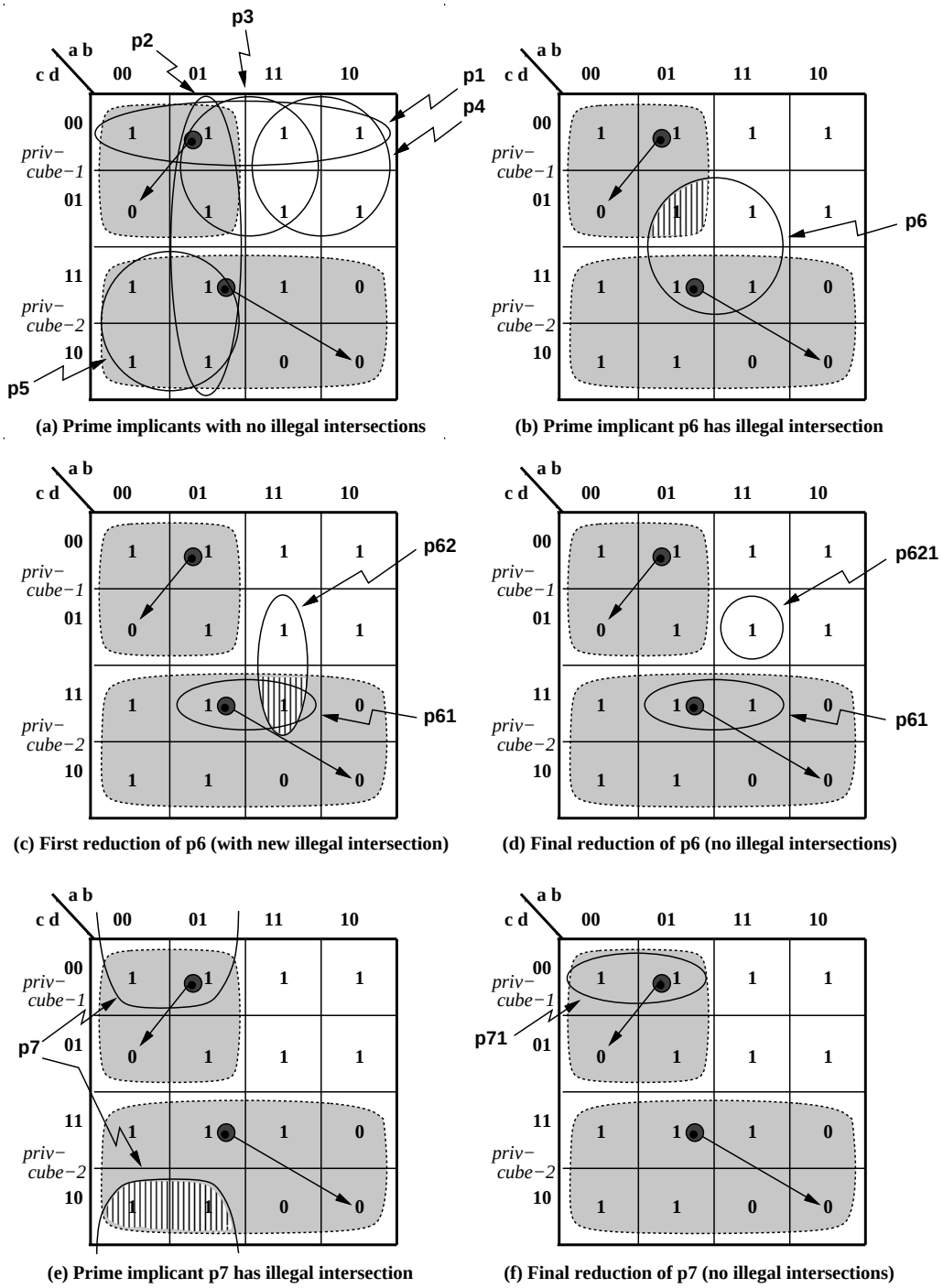


Figure 5: Hazard-Free Minimization Example: **Step 1**

	required cubes				
dhf-prime implicants	ac'	$a'c'd'$	$a'bc'$	$a'c$	bcd
p_1=$c'd'$		X			
p_2=$a'b$			X		
p_3=bc'			X		
p_4=ac'	X				
p_5=$a'c$				X	
p_{61}=bcd					X

Table 3: Hazard-Free Minimization Example: **Step 2**.

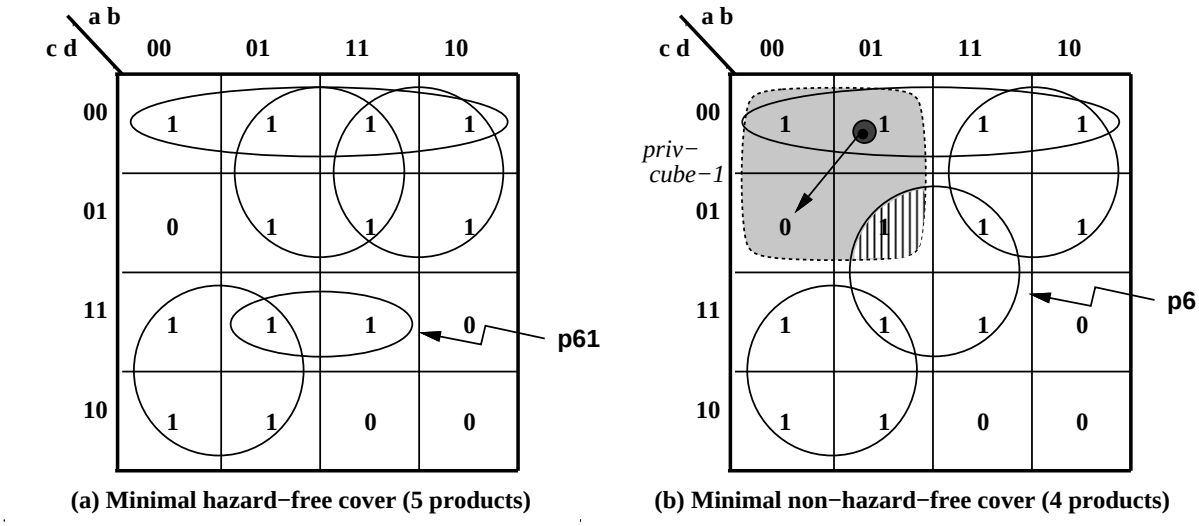
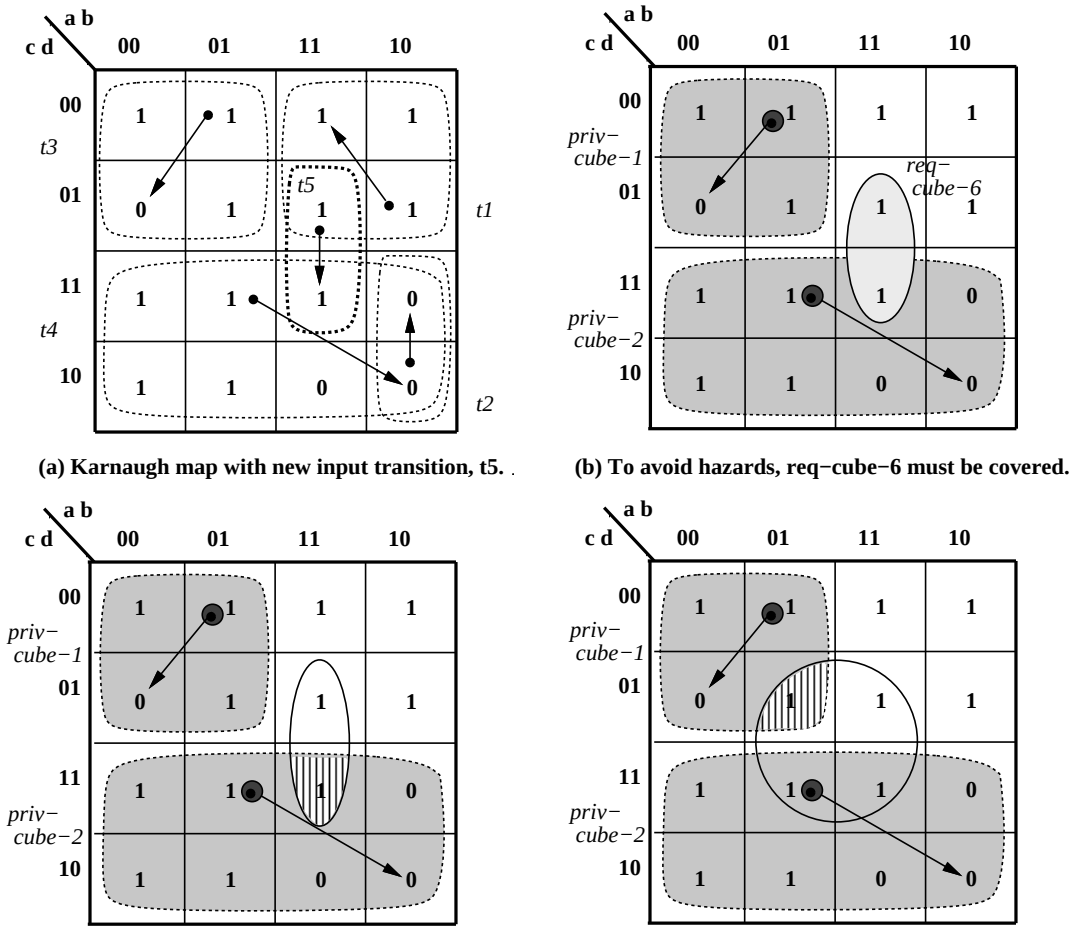


Figure 6: Hazard-Free Minimization Example: **Step 3**



(a) Karnaugh map with new input transition, t5.

(b) To avoid hazards, req−cube−6 must be covered.

(c) and (d): Every implicant which covers req−cube−6 has an illegal intersection.

Figure 7: Boolean Function with No Hazard-Free Cover

Example. We consider the function used in the previous section, but augment its set $T = \{t_1, t_2, t_3, t_4\}$ of specified input transitions with a new transition:

$$t_5\text{:} \quad m_5 = abc'd \qquad C_5 = abd$$

The input transitions are indicated in the Karnaugh map of figure 7(a). The req-set now has an additional required cube: *req-cube-6* = *abd*. The off-set and priv-set are unchanged from the example of Section 7, and the function has the same dhf-prime implicants as well.

Figure7(b)-(d) illustrates the covering problem. To insure no static hazard for transition *t5*, the required cube *req-cube-6* must be covered by some product. However, every product which contains *req-cube-6* also illegally intersects a privileged cube, *priv-cube-1* or *priv-cube-2*. That is, any attempt to eliminate the static hazard in transition t_5 will produce a dynamic hazard in one of the transitions, t_3 or t_4.

Table 4 shows the resulting dhf-prime implicant table. This table has no solution: no dhf-prime implicant contains required cube *abd*.

	required cubes					
dhf-prime implicants	ac'	$a'c'd'$	$a'bc'$	$a'c$	bcd	abd
p_1=$c'd'$		X				
p_2=$a'b$			X			
p_3=bc'			X			
p_4=ac'	X					
p_5=$a'c$				X		
p_{61}=bcd					X	

Table 4: DHF-Prime Implicant Table Having No Solution.

9 Comparison with Frackowiak's Work.

It is useful to compare our approach with the related work of Frackowiak [12]. Frackowiak presents two hazard-free minimization algorithms for two-level implementations allowing multiple-input changes. The algorithms assume that functions are fully-specified.

Both algorithms eliminate dynamic hazards in specified transitions. However, the first method (*Algorithm A*) ignores static hazards. The second method (unnamed, but here called *Algorithm A'*) attempts to eliminate static hazards for *every* static transition, even if unspecified. Therefore results may be either hazardous (Algorithm A) or suboptimal (Algorithm A').

In particular, Algorithm A first generates all dhf-prime implicants, then attempts to cover every ON-set minterm (not required cube) using a dhf-prime implicant. The algorithm finds a minimal cover which is hazard-free for a given set of dynamic transitions, if a solution exists. Since required cubes are not covered, static hazards may occur.

Algorithm A' attempts to eliminate both dynamic and static hazards. The algorithm extends an earlier result by Eichelberger [11]. Eichelberger proved that, to eliminate all static logic hazards for a fully-specified function, a cover must include all prime implicants. Frackowiak's goal is different: first, to eliminate hazards for a given set of dynamic transitions and, second, to eliminate as many static hazards as possible. Therefore, his solution is *to include all dhf-prime implicants* in the cover. Since only dhf-primes are used, every specified dynamic transition will be hazard-free (if a hazard-free solution exists). Furthermore, by using all dhf-prime implicants, as many remaining static hazards as possible are eliminated.

Algorithm A' and our algorithm are both guaranteed to find a hazard-free cover, if one exists. However, since Algorithm A' includes all dhf-prime implicants in the solution, the resulting cover may be far from minimal. In fact, judging from the non-hazard-free case [32], the number of primes for even small examples may be huge; in this case, Algorithm A' will not be practical. In contrast, our algorithm finds a minimum-cost hazard-free solution.

Example. The Karnaugh map of figure 8(a) describes a fully-specified Boolean function. The function has four specified input transitions. Each transition t_i is described by its transition cube C_i and start point m_i:

$$\begin{array}{lll} t_1: & m_1 = a'bc'd' & C_1 = a'c' \\ t_2: & m_2 = a'b'cd' & C_2 = c \\ t_3: & m_3 = a'b'cd' & C_3 = a'd' \\ t_4: & m_4 = ab'c'd' & C_4 = ac' \end{array}$$

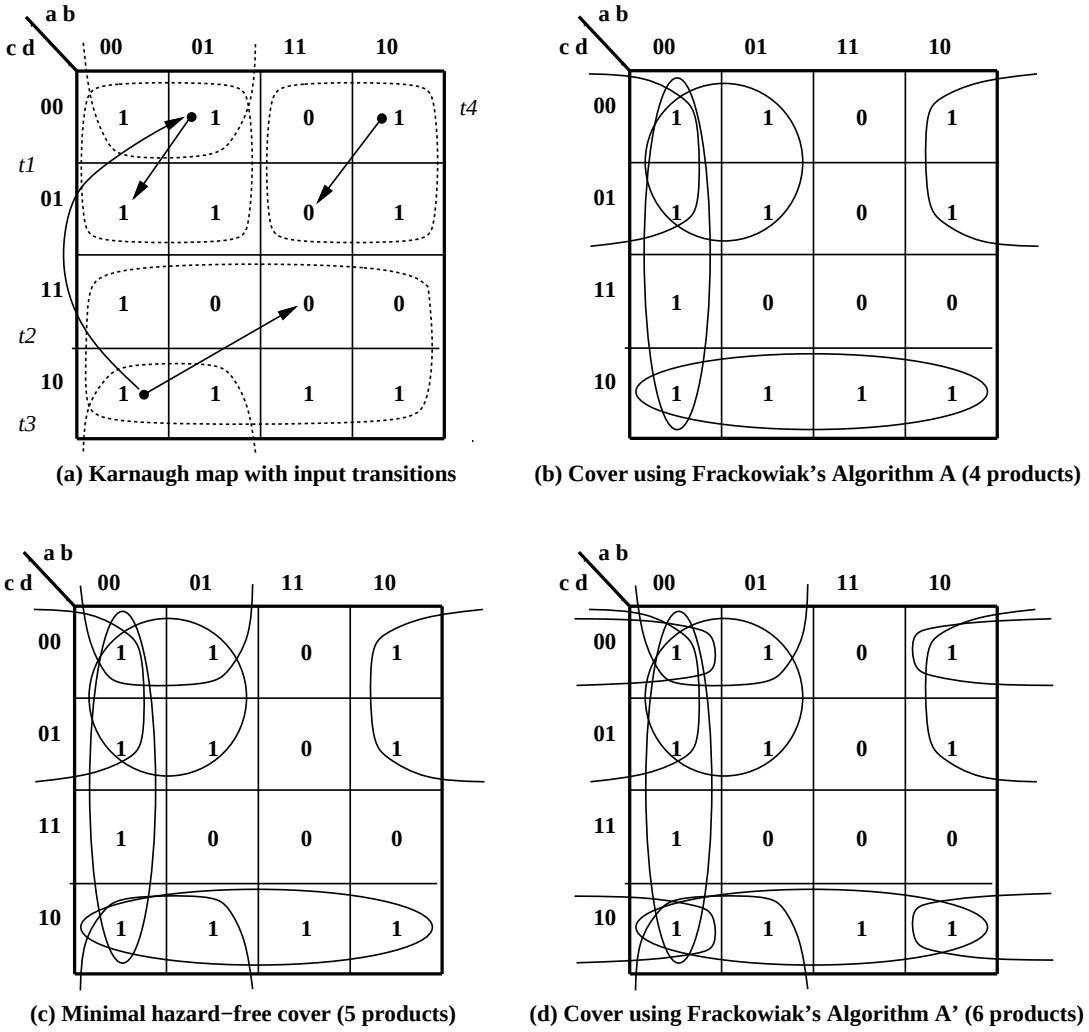


Figure 8: Comparison with Frackowiak's Method

A minimal cover using Frackowiak's Algorithm A has 4 products (see figure 8(b)). It is hazard-free for dynamic transitions t_2 and t_4, but has a static logic hazard for transition t_3.

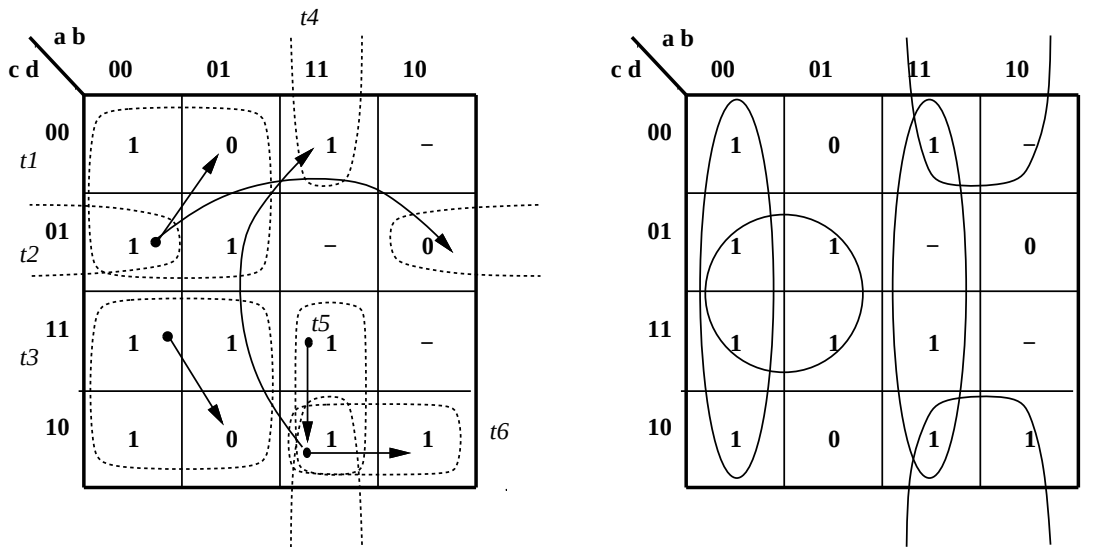
A minimal hazard-free cover, using our method, is shown in figure 8(c). The cover has 5 products and is hazard-free for every specified transition.[2]

Finally, a minimal cover using Frackowiak's Algorithm A' is shown in figure 8(d). The cover is hazard-free for every specified transition but has 6 products; it is therefore suboptimal. □

A final distinction between our work and Frackowiak, is that we allow incompletely-specified functions:

Example. The Karnaugh map of figure 9(a) describes an incompletely-specified Boolean function. The function has six specified input transitions:

[2]Interestingly, this solution is prime but redundant, since it contains prime implicant *a'd'*. In contrast, the solution for the previous example (figure 6(a)) was non-prime but irredundant.



(a) Karnaugh map with input transitions **(b) Minimal hazard−free cover (4 products)**

Figure 9: Hazard-Free Minimization of an Incompletely-Specified Boolean Function

$$
\begin{array}{lll}
t_1\text{:} & m_1 = a'b'c'd & C_1 = a'c' \\
t_2\text{:} & m_2 = a'b'c'd & C_2 = b'c'd \\
t_3\text{:} & m_3 = a'b'cd & C_3 = a'c \\
t_4\text{:} & m_4 = abcd' & C_4 = abd' \\
t_5\text{:} & m_5 = abcd & C_5 = abc \\
t_6\text{:} & m_6 = abcd' & C_6 = acd'
\end{array}
$$

A minimal cover, using our method, is shown in figure 9(b). The cover has 4 products and is hazard-free for every specified input transition.

10 Program Implementation.

We have implemented the logic minimization algorithms of the Section 6. Our program is written in Lucid Common Lisp and is run on a DECStation 3100. However, it makes use of *espresso* [5, 33] to perform part of its computation: prime implicant generation. The advantage of this approach is that we can benefit from highly optimized existing tools.

The program generates sets for a function (*Step 0*) and writes the ON-set and OFF-set into a file in PLA format. We then use *espresso -Dprimes* to generate all prime implicants. The resulting PLA file is read in by the program, which computes the sets of dhf-prime implicants (*Step 1*). The program then constructs a dhf-prime implicant table and solves it (*Steps 2* and *3*).

This logic minimization program has been used as the the final component in an existing synthesis program for asynchronous controllers [25, 28]. It has recently been incorporated into two other asynchronous synthesis programs as well [38, 26]. All three methods produce combinational functions which are guaranteed by construction to have hazard-free two-level implementations. In particular, each method imposes constraints during state minimization to insure that hazard-free covers will exist for the resulting Boolean functions (for a detailed discussion, see [25]).

		prime implicants		dhf-prime implicants	
name	in/out	total	% illegal	total	% non-prime
dean-ctrl	20/19	1676	4	997	7
oscsci-ctrl	14/5	192	3	140	2
scsi-ctrl	12/5	280	1	190	2
pe-send-ifc	7/3	22	5	20	5
chu-ad-opt	4/3	6	0	4	0
vanbek-opt	4/3	7	0	6	0
dme	5/3	9	0	6	0
dme-opt	5/3	7	0	6	0
dme-fast	5/3	10	0	7	0
dme-fast-opt	5/3	15	0	14	0

Table 5: Results of Algorithm *PI-to-DHF-PI*.

11 Experimental Results.

Our hazard-free logic minimization program was run on a set of examples. The largest example is a cache controller having 20 inputs and 19 outputs (*dean-ctrl*) [27]. The program was also run on two SCSI controller designs (*oscsi-ctrl* and *scsi-ctrl*) [31]. The examples were generated from state machine specifications using the locally-clocked synthesis method [25]. Specifications were given in "burst-mode" [29, 25], a notation to describe asynchronous Mealy machines allowing multiple-input changes. Several examples have appeared previously in the literature using other concurrent description languages (STGs [10, 36], CSP [9]); reasonable timing assumptions were made when synthesizing them as multiple-input-change state machines (see [28]).

Table 5 describes the results of Algorithm *PI-to-DHF-PI*. The algorithm transforms prime implicants into dhf-prime implicants. Prime implicants which contain only don't-care minterms are not included, since these implicants will never appear in an exact solution.

Illegal prime implicants are those which are illegally intersect some privileged cube, and therefore are not dhf-prime implicants. In every case, no more than 5% of the original prime implicants are illegal and must be further reduced.

After reduction, at most 7% of the dhf-prime implicants are not prime. It is also interesting that a number of prime implicants are discarded by the algorithm (see *dean-ctrl*). These implicants contain ON-set minterms but contain no required cubes. Since these implicants do not contribute to the hazard-free covering solution, they can be removed.

Table 6 presents the exact hazard-free solutions for the examples. It also gives an indication of the penalty associated with hazard elimination in our algorithms. In every case, the overhead for hazard-elimination is no more than a 6% increase in the number of products as compared with outputs synthesized using *espresso-exact*.

Runtimes were quite reasonable for all examples tested. Even for the cache controller example, with 20 inputs and 19 outputs, total runtime was 83 seconds.

		Total Products			
name	in/out	Hazard-free Method	espresso-exact	% Over-head	Hazard-free Run-time(s)
dean-ctrl	20/19	215	202	6	83
oscsci-ctrl	14/5	59	58	2	9
scsi-ctrl	12/5	60	59	2	11
pe-send-ifc	7/3	15	15	0	1
chu-ad-opt	4/3	4	4	0	1
vanbek-opt	4/3	6	6	0	1
dme	5/3	4	4	0	1
dme-opt	5/3	4	4	0	1
dme-fast	5/3	5	5	0	1
dme-fast-opt	5/3	8	8	0	1

Table 6: Comparison of Hazard-Free Logic Minimization with *espresso-exact*.

12 Conclusions.

This paper considers the two-level hazard-free minimization problem for several reasons: the general problem has not previously been solved; minimal two-level solutions are important for optimal PLA implementations; and solutions serve as a good starting point for hazard-preserving multi-level logic transformations. In particular, multi-level transformations which introduce no hazards into a combinational network are discussed in [35]. This set of transformations has been significantly extended by Kung [14]. Finally, technology mapping algorithms which introduce no hazards are described by Siegel *et al.* [34].

We have described the problem of implementing hazard-free two-level logic as a constrained covering problem on Karnaugh maps. We presented an automated algorithm for solving the two-level hazard-free logic minimization problem and showed its effectiveness on a set of examples.

An important feature of the algorithms is that they involve only *localized* changes to existing algorithms. As a result, we can use existing sophisticated algorithms for prime implicant generation (Step 1) and for table reduction and solution (Step 3).

Our algorithms have implications for testability, since they may introduce redundant and non-prime implicants. In this case, the resulting circuits may have non-testable faults. However, recent methods have been proposed which insure complete testability of hazard-free logic, for both stuck-at and robust path delay faults, in the presence of redundant [13, 30] and non-prime [30] implicants. Therefore, testability need not be adversely affected when hazards are removed.

With the automation of these exact algorithms, the basic automated synthesis system of [28] is complete. The algorithms have been incorporated into two other synthesis systems as well [38, 26] and can be used in a number of other sequential synthesis methods. The algorithms have also been applied to several substantial asynchronous designs, including a second-level cache controller [27] and state machines for an infrared communications chip [1].

Acknowledgements.

The authors would like to thank Professor Giovanni De Micheli for suggesting that we consider reducing prime implicants to dhf-prime implicants. We thank Richard Rudell for clarifications about espresso-exact and Kenneth Yun for interesting discussions and examples.

References

[1] B. Coates A. Marshall and P. Siegel. The design of an asynchronous communications chip. *Design and Test*, June 1994.

[2] V. Akella and G. Gopalakrishnan. SHILPA: a high-level synthesis system for self-timed circuits. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 587–91, November 1992.

[3] P.A. Beerel and T. Meng. Automatic gate-level synthesis of speed-independent circuits. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 581–586, November 1992.

[4] J. Beister. A unified approach to combinational hazards. *IEEE Transactions on Computers*, C-23(6), 1974.

[5] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Boston, MA, 1984.

[6] J.G. Bredeson. Synthesis of multiple input-change hazard-free combinational switching circuits without feedback. *Int. J. Electronics*, 39(6):615–624, 1975.

[7] J.G. Bredeson and P.T. Hulina. Elimination of static and dynamic hazards for multiple input changes in combinational switching circuits. *Information and Control*, 20:114–224, 1972.

[8] E. Brunvand and R. F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *IEEE International Conference on Computer-Aided Design*, pages 262–265, November 1989.

[9] S. M. Burns. Automated compilation of concurrent programs into self-timed circuits. Technical Report Caltech-CS-TR-88-2, California Institute of Technology, 1987. M.S. Thesis.

[10] T.-A. Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. Technical Report MIT-LCS-TR-393, Massachusetts Institute of Technology, 1987. Ph.D. Thesis.

[11] E.B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM J. Res. Develop.*, 9(2):90–99, 1965.

[12] J. Frackowiak. Methoden der analyse und synthese von hasardarmen schaltnetzen mit minimalen kosten I. *Elektronische Informationsverarbeitung und Kybernetik*, 10(2/3):149–187, 1974.

[13] K. Keutzer, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis for testability techniques for asynchronous circuits. In *IEEE International Conference on Computer-Aided Design*, pages 326–329, November 1991.

[14] D.S. Kung. Hazard-non-increasing gate-level optimization algorithms. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 631–634, November 1992.

[15] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *ACM/IEEE Design Automation Conference*, pages 302–308, June 1991.

[16] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1:226–234, 1986.

[17] H.J. Mathony. A universal logic design algorithm and its application to the synthesis of two-level switching circuits. *IEE Proceedings*, 136(3):171–177, May 1989.

[18] E.J. McCluskey. *Introduction to the Theory of Switching Circuits.* McGraw-Hill, New York, NY, 1965.

[19] E.J. McCluskey. *Logic Design Principles: with emphasis on testable semicustom circuits.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[20] P.C. McGeer and R.K. Brayton. Hazard prevention in combinational circuits. In *Hawaii International Conference on System Sciences*, volume I, pages 111–120, January 1990.

[21] R.B. McGhee. Some aids to the detection of hazards in combinational switching circuits. *IEEE Transactions on Computers (Short Notes)*, C-18:561–565, June 1969.

[22] T. H.-Y. Meng, R.W. Brodersen, and D.G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design*, 8(11):1185–1205, November 1989.

[23] C.W. Moon, P.R. Stephan, and R.K. Brayton. Synthesis of hazard-free asynchronous circuits from graphical specifications. In *IEEE International Conference on Computer-Aided Design*, pages 322–325, November 1991.

[24] C. Myers and T. Meng. Synthesis of timed asynchronous circuits. In *IEEE International Conference on Computer Design*, pages 279–284, October 1992.

[25] S.M. Nowick. Automatic synthesis of burst-mode asynchronous controllers. Technical report, Stanford University, 1993. Ph.D. Thesis.

[26] S.M. Nowick and B. Coates. UCLOCK: automated design of high-performance unclocked state machines. In *IEEE International Conference on Computer Design*, October 1994.

[27] S.M. Nowick, M.E. Dean, D.L. Dill, and M. Horowitz. The design of a high-performance cache controller: a case study in asynchronous synthesis. *INTEGRATION, the VLSI journal*, 15(3):241–262, October 1993.

[28] S.M. Nowick and D.L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *IEEE International Conference on Computer-Aided Design*, pages 318–321, November 1991.

[29] S.M. Nowick and D.L. Dill. Synthesis of asynchronous state machines using a local clock. In *IEEE International Conference on Computer Design*, pages 192–197, October 1991.

[30] S.M. Nowick, N.K. Jha, and F.-C. Cheng. Synthesis of asynchronous circuits for stuck-at and robust path delay fault testability. In *Eighth International Conference on VLSI Design*, January 1995.

[31] S.M. Nowick, K.Y. Yun, and D.L. Dill. Practical asynchronous controller design. In *IEEE International Conference on Computer Design*, pages 341–345, October 1992.

[32] R. Rudell. Logic synthesis for VLSI design. Technical Report UCB/ERL M89/49, Berkeley, 1989. Ph.D. Thesis.

[33] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minmization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, 6(5):727–750, September 1987.

[34] P. Siegel, G. De Micheli, and D. Dill. Technology mapping for generalized fundamental-mode asynchronous designs. In *ACM/IEEE Design Automation Conference*, June 1993.

[35] S.H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, New York, NY, 1969.

[36] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *IEEE International Conference on Computer-Aided Design*, pages 184–187, November 1990.

[37] M.L. Yu and P.A. Subrahmanyam. A path-oriented approach for reducing hazards in asynchronous designs. In *IEEE/ACM Design Automation Conference*, pages 239–244, June 1992.

[38] K.Y. Yun, D.L. Dill, and S.M. Nowick. Synthesis of 3D asynchronous state machines. In *IEEE International Conference on Computer Design*, pages 346–350, October 1992.